Contents

[REACT JS 4](#_Toc171286164)

[WHAT IS REACT? 4](#_Toc171286165)

[REACT CONCEPTS AND CREATING REACT COMPONENTS 4](#_Toc171286166)

[BASIC STEPS TO CREATE REACT COMPONENT 5](#_Toc171286167)

[UNDERSTANDING JSX 5](#_Toc171286168)

[JSX COMPILATION 5](#_Toc171286169)

[HTML TO JSX 6](#_Toc171286170)

[ADDING CSS LIBRARY WITH NPM 6](#_Toc171286171)

[ADDING CUSTOM CSS 6](#_Toc171286172)

[JSX REFERING JS VARIABLES 7](#_Toc171286173)

[JSX RESTICTIONS 7](#_Toc171286174)

[REACT FRAGMENT 7](#_Toc171286175)

[IMPORTING IMAGES 8](#_Toc171286176)

[PROPS 8](#_Toc171286177)

[VALUE TYPES IN PROPS 8](#_Toc171286178)

[PASSING VALUE USING PROPS 9](#_Toc171286179)

[ALTERNATIVE PROPS SYNTAX 9](#_Toc171286180)

[PASSING A SINGLE PROP OBJECT 10](#_Toc171286181)

[GROUPING RECEIVED PROPS INTO A SINGLE OBJECT 10](#_Toc171286182)

[DEFAULT PROP VALUES 10](#_Toc171286183)

[children PROP 11](#_Toc171286184)

[CREATING A DEMO REACT APP – BLOG POST APP 12](#_Toc171286185)

[COMMUNICATION BETWEEN COMPONENTS – PARENT TO CHILD 13](#_Toc171286186)

[HANDLING EVENTS AND USER INPUTS 14](#_Toc171286187)

[BINDING EVENTS 14](#_Toc171286188)

[CHILD TO PARENT COMMUNICATION 15](#_Toc171286189)

[MANAGING STATES 15](#_Toc171286190)

[REACT HOOKS 15](#_Toc171286191)

[BUILT IN HOOKS 15](#_Toc171286192)

[RULES OF HOOKS 16](#_Toc171286193)

[STATES IN COMPONENTS 16](#_Toc171286194)

[CHILD TO PARENT COMMUNICATION 18](#_Toc171286195)

[CONDITIONS 18](#_Toc171286196)

[STATE UPDATE 18](#_Toc171286197)

[ADDING ELEMENT 18](#_Toc171286198)

[REMOVING ELEMENT 19](#_Toc171286199)

[CHANGING ELEMENT 19](#_Toc171286200)

[CHANGING PROPERTIES 19](#_Toc171286201)

[REMOVING PROPERTIES 20](#_Toc171286202)

[APPLYING DYNAMIC CSS CLASS 20](#_Toc171286203)

[FORWARDING PROPS 20](#_Toc171286204)

[TWO WAY DATA BINDING 21](#_Toc171286205)

[HTTP REQUEST IN REACT 21](#_Toc171286206)

[APP OVERVIEW- SEARCH IMAGES 21](#_Toc171286207)

[DATA PERSISITENCE 23](#_Toc171286208)

[SETTING UP JSON SERVER 23](#_Toc171286209)

[WORKING WITH JSON-SERVER 24](#_Toc171286210)

[USING REST CLIENT 25](#_Toc171286211)

[UPDATING THE APP TO INTERACT WITH JSONSERVER 26](#_Toc171286212)

[useEffect REACT HOOK 27](#_Toc171286213)

[COMMUNICATION WITH CONTEXT SYSTEM 27](#_Toc171286214)

[SETTING UP THE CONTEXT 28](#_Toc171286215)

[SIMPLE CONTEXT EXAMPLE 28](#_Toc171286216)

[HOW TO UPDATE THE VALUE IN CONTEXT? 29](#_Toc171286217)

[EXAMPLE 30](#_Toc171286218)

[useReducer 30](#_Toc171286219)

[CONCEPT 32](#_Toc171286220)

[RULES OF REDUCER FUNCTION 32](#_Toc171286221)

[ACTION OBJECT 32](#_Toc171286222)

[EXAMPLE “useReducer” 33](#_Toc171286223)

[IMMER 35](#_Toc171286224)

[IMMER IN ACTION 35](#_Toc171286225)

[CREATING REUSABLE COMPONENTS 35](#_Toc171286226)

[REUSABLE BUTTON COMPONENT 35](#_Toc171286227)

[VALIDATION 36](#_Toc171286228)

[REDUX 40](#_Toc171286229)

[APP OVERVIEW – USING REDUX 41](#_Toc171286230)

[UNDERSTANDING STORE 43](#_Toc171286231)

[UNDERSTANDING SLICES 43](#_Toc171286232)

[UNDERSTANDING ACTION CREATORS 44](#_Toc171286233)

[CONNECTING REACT TO REDUX 45](#_Toc171286234)

[STEPS TO CONNECT REDUX TO REACT 45](#_Toc171286235)

[ACCESSING STORE IN COMPONENT 46](#_Toc171286236)

[STEPS 46](#_Toc171286237)

[UPDATING THE COMPONENT WITH STATE 48](#_Toc171286238)

[SELECTOR FUNCTION 48](#_Toc171286239)

[RESETING THE STATE 49](#_Toc171286240)

# REACT JS

## WHAT IS REACT?

* React is JS library which is used to create user interfaces. It has component-based architecture.
* As the name suggest – Reacts to the state change.

|  |  |  |
| --- | --- | --- |
| **INSTALLING REACT** | npm install -g create-react-app | |
| **CREATING REACT PROJECT** | npx create-react-app <project\_name>  npx create-react-app react-complete-app |  |
| **STARTING NPM SERVER** | cd react-complete-app  npm start |
| **CHANGING THE DEFAULT PORT** | $env:PORT=5000  npm start | |
|  | We can run the command to intstall react globally and create the react app - in one go.  “npx” only works with npm version > 5.2 | |

## REACT CONCEPTS AND CREATING REACT COMPONENTS

|  |  |  |  |
| --- | --- | --- | --- |
|  | | * A typical React app could be depicted as a component tree - having one root component ("App") and then a potentially infinite amount of nested child components. * When we create a react application, we build bunch of isolated, independent, and reusable components, which then composed together to build complex component. * Every React Application has once parent component called root component called “App” component. | |
|  | | * Every component in react is technically a JS class, which has * State 🡪 The describes the state of the UI component when it will load on the page * render() 🡪 This describes how the UI of the component will look like | |
|  | | * The react element/component is a JS object which is directly mapped with the DOM in the browser * React keep the lightweight representation of the DOM Elements in memory called Virtual DOM * When we update any react component - to keep the DOM in sync (to match the state) , it will only update that part of DOM which is mapped to that component | |
|  |  | | * The node\_modules for has all the dependencies, which we add in package.json file as a dependency. * In public folder has one index.html file which is served by the webserver. This is html file where all our react script will be injected (App root component js). * It has one root div which will become the parent of the react app * For SPA we don’t edit this html, because the root div is the place holder where we will mount our react app * We still can add CSS files to this file * <div id="root"></div> |

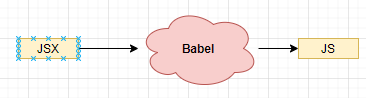
|  |
| --- |
| **src** - This folder has an App component which will be the parent component of all custom component we will create a in react manifest.json file give the basic functionality of PWA to react app. |

## BASIC STEPS TO CREATE REACT COMPONENT

|  |  |
| --- | --- |
| **IMPORT REACT AND REACT DOM LIBRARY** | import React from 'react';  import ReactDOM from 'react-dom'; |
| **CREATE A REACT COMPONENT** | function App {  return (  <h1>Returning JSX</h1> 🡨 JSX expression  );  }  export default App; |
| **RENDER THE COMPONENT IN THE BROWSER** | ReactDOM.render(<App /> ,document.getElementById('root')); |
|  | **OVERVIEW**   * Each component needs to return/ render some JSX code - it defines which HTML code React should render to the real DOM in the end. * JSX is NOT HTML but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls. * The components we create are injected in App component (as a child component) and finally the App component script is injected in index.html file. * Hence the components will be child of App component. * The class based react component can be created by extending Component * It has render method which has to return valid JSX expression (JSX looks very similar to HTML, but it’s not) * The JSX expression then gets compiled into JS (its equivalent createElement() method) |

## UNDERSTANDING JSX

* The JSX expression looks very much like HTML. The JSX expression is finally compiled to JS [<https://babeljs.io/repl>]



### JSX COMPILATION

|  |  |  |  |
| --- | --- | --- | --- |
| **JSX** | | **COMPILED JSX** | |
| <h1 className="heading">Hello</h1> | | React.createElement("h1", { className: "heading"}, "Hello"); | |
| <p className="heading"><span className="spanStyle">Hello</span></p> | | "use strict";  React.createElement("p", { className: "heading" }, React.createElement("span", { className: "spanStyle"}, "Hello")); | |
| * The JSX expression compiled as a JS code. The compiled JSX has createElement method which created the DOM elements in the browser. So, we can use React.createElement method too to render the HTML element. * ***We use “className” instead class because the class is a reserved word in JS*** | | | |
| **UNDERSTANDING React.createElement METHOD** | | | |
| EXAMPLE: - **React.createElement('h1', {}, 'My First React Code');** | | | |
| * The first one is the type of element we're creating, in this case an <h1> tag. This could also be another React component. If we're creating an HTML element, we pass in the name as a string, just like we did above. If we're creating a React component, we pass in the variable that the component is assigned to. * The second argument is an object containing properties ('props' in React terms) that get passed to the component. * Finally, the last argument is the children of that component. This can be a quoted string like shown above, in which case the content will be interpreted as text. However, we can also pass in a reference to another component, allowing us to nest elements and components within each other (we'll get to that in a bit). * To render the component. We do this using ReactDOM.render(). This takes two arguments: the first one being the thing we want to render (our title element), and the second one is a target DOM node to render things into. | | | |
| **APP JS** | **INDEX.JS** | | **INDEX.HTML** |
| import React, { Component } from "react";  class App extends Component {  **render**() {  **return** React.createElement("div", { className: "heading" }, "Test Data");  }  }  export default App; | import React from 'react';  import ReactDOM from 'react-dom';  import App from './App';  ReactDOM.render(<App />, document.getElementById('root')); | | <div id="root">  </div> |
|  | | |

### HTML TO JSX

#### RULES

1. **All prop names should follow camelCase**
2. **Number attributes use curly braces**
3. **Boolean true can be written with just the property name. ‘False’ should be written with curly braces.**
4. **The “class” attribute is written as ‘className’**
5. **In-line style are provided as an object**

**EXAMPLE (RULE 1 & 2)**

|  |  |
| --- | --- |
| **HTML** | **EQUIVALENT JSX** |
| <input maxlength=”5”/> | <input maxLength={5}/> |
| <form autocapitialize/> | <form autoCapitialize/> |
| <form novalidate/> | <form noValidate/> |

**EXAMPLE (RULE 3)**

|  |  |
| --- | --- |
| **HTML** | **EQUIVALENT JSX** |
| <input spellcheck=”true”/> | <input spellCheck/> |
| <input spellcheck=”false”/> | <input spellCheck={false}/> |

**EXAMPLE (RULE 5)**

|  |  |
| --- | --- |
| <a style=”text-decoration:’none’; padding:’5px’;”/> | <a style**={{textDecoration:’none’,padding:’5px’}}/**> |

### ADDING CSS LIBRARY WITH NPM

|  |  |
| --- | --- |
| * Let’s say we are going to use “**bulma**” css library in our project- <https://bulma.io/> * INSTALL BULMA - **npm install bulma** | import "bulma/css/bulma.css";  function App() {  return (  <section className="hero is-primary">  ....  </section>  </div>  );  }  export default App; |

### ADDING CUSTOM CSS

|  |  |
| --- | --- |
|  | function App() {  return(  ...JSX  )  }  export default App; |

### JSX REFERING JS VARIABLES

|  |  |
| --- | --- |
| **JSX CAN ALSO REFER THE JAVASCRIPT VARIABLE** | const App = () => {  const message = "Hello World";  return <div><h1 >{message}</h1> </div>;  }; |
| **JSX CAN ALSO REFER THE JAVASCRIPT FUNCTIONS** | function getMessage() {  return "Hello World!";  }  const App = () => {  return <div><h1>{getMessage()}</h1> </div>;  }; |

### JSX RESTICTIONS

* All prop names follow camelCase
* Number attributes use curly braces.
* Boolean “true” can be written with just property name. ‘False’ should be written with curly braces
* Inline-styles are provided as objects
* We should not use JavaScript reserved word. For example, we use “className” instead class, because the class is a reserved word in JS.
* There should be one parent element in a JSX expression. But there is a way to overcome this limitation

|  |  |
| --- | --- |
| **VALID** | **INVALID** |
| <div>  <p>Hello</p>  <p>World</p>  </div> | <div>  <p>Hello</p>  </div>  <p>World</p> |

### REACT FRAGMENT

* React Fragment is a built-in component in React that allows you to group multiple elements together without adding an extra DOM node. It is useful when you need to return multiple elements from a component without adding an unnecessary wrapper element.
* In JSX, when you have multiple elements that need to be rendered together, you typically wrap them in a parent element, like a <div>. However, this can sometimes lead to unnecessary nesting, especially when you don't need the extra <div> for styling or other purposes. React Fragment provides a solution to this problem.

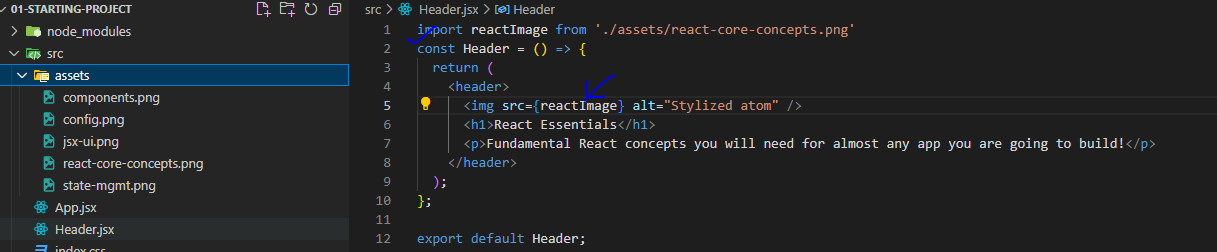
EXAMPLE

|  |
| --- |
| import React from 'react';  function MyComponent() {  return (  <React.Fragment>  <p>First paragraph</p>  <p>Second paragraph</p>  </React.Fragment>  );  } |

You can also use a shorthand syntax for React Fragment, which is an empty tag <> and </>:

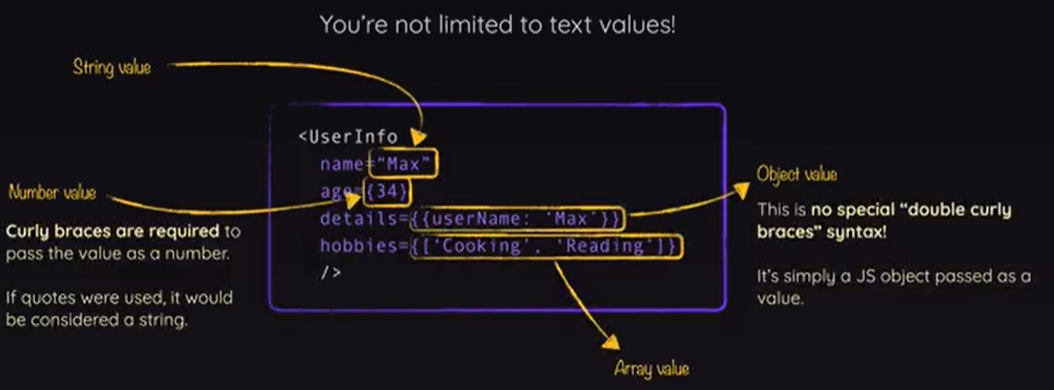
|  |
| --- |
| import React from 'react';  function MyComponent() {  return (  **<>**  <p>First paragraph</p>  <p>Second paragraph</p>  **</>**  );  } |

## IMPORTING IMAGES

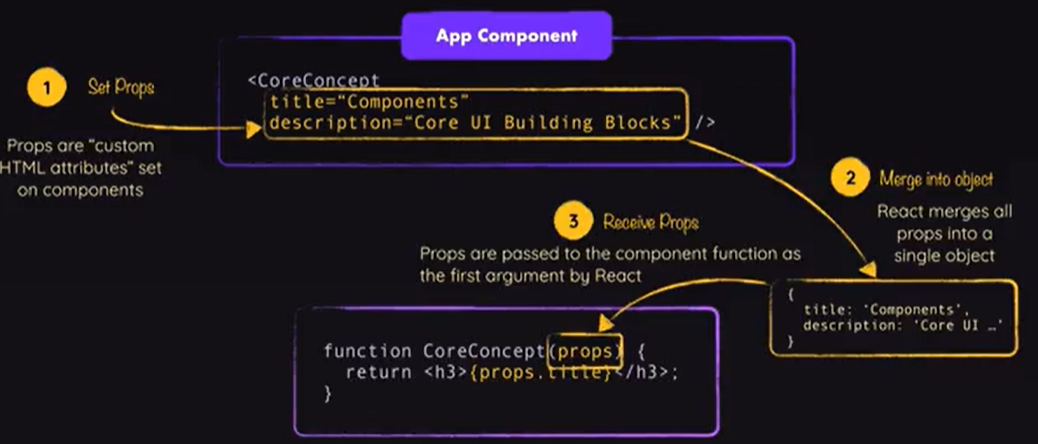


## PROPS

### VALUE TYPES IN PROPS



### PASSING VALUE USING PROPS



### ALTERNATIVE PROPS SYNTAX

|  |
| --- |
| App.jsx |
| import CoreConcept from "./CoreConcept";  import Header from "./Header";  import componentImage from "./assets/components.png";  import { CORE\_CONCEPTS } from "./data";  function App() {  const coreConcepts = CORE\_CONCEPTS.map((concept, index) => (  <CoreConcept key={index} title={concept.title} description={concept.description} image={concept.image} />  ));  return (  <div>  <Header />  <main>  <section id="core-concepts">  <ul>  {coreConcepts}  </ul>  </section>  </main>  </div>  );  }  export default App; |

|  |  |
| --- | --- |
| VARIATION-1 | VARIATION 2 |
| const CoreConcept = (props) => {  const { title, description, image } = props;  return (  <li>  <img src={image} alt="Core Concept" />  <h3>{title}</h3>  <p>{description}</p>  </li>  );  };  export default CoreConcept; | const CoreConcept = ({ title, description, image }) => {  return (  <li>  <img src={image} alt="Core Concept" />  <h3>{title}</h3>  <p>{description}</p>  </li>  );  };  export default CoreConcept; |

### **PASSING A SINGLE PROP OBJECT**

If we got data that's already organized as a JavaScript object, you can pass that object as a single prop value instead of splitting it across multiple props. I.e., instead of

<CoreConcept

title={CORE\_CONCEPTS[0].title}

description={CORE\_CONCEPTS[0].description}

image={CORE\_CONCEPTS[0].image} />

or

<CoreConcept

{...CORE\_CONCEPTS[0]} />

We could also pass a single concept (or any name of your choice) prop to the CoreConcept component:

1. <CoreConcept
2. concept={CORE\_CONCEPTS[0]} />

In the CoreConcept component, you would then get that one single prop:

export default function CoreConcept({ concept }) {

// Use concept.title, concept.description etc.

// Or destructure the concept object: const { title, description, image } = concept;

}

### **GROUPING RECEIVED PROPS INTO A SINGLE OBJECT**

You can also pass multiple props to a component and then, in the component function, group them into a single object via JavaScript's ["Rest Property"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#rest_property) syntax.

I.e., if a component is used like this:

<CoreConcept

title={CORE\_CONCEPTS[0].title}

description={CORE\_CONCEPTS[0].description}

image={CORE\_CONCEPTS[0].image} />

**You could group the received props into a single object like this:**

export default function CoreConcept({ ...concept }) {

// ...concept groups multiple values into a single object

// Use concept.title, concept.description etc.

// Or destructure the concept object: const { title, description, image } = concept;

}

### **DEFAULT PROP VALUES**

Sometimes, you'll build components that may receive an optional prop. For example, a custom Button component may receive a type prop.

So the Button component should be usable either with a type being set:

<Button type="submit" caption="My Button" />

Or without it:

<Button caption="My Button" />

To make this component work, you might want to set a default value for the type prop - in case it's not passed.

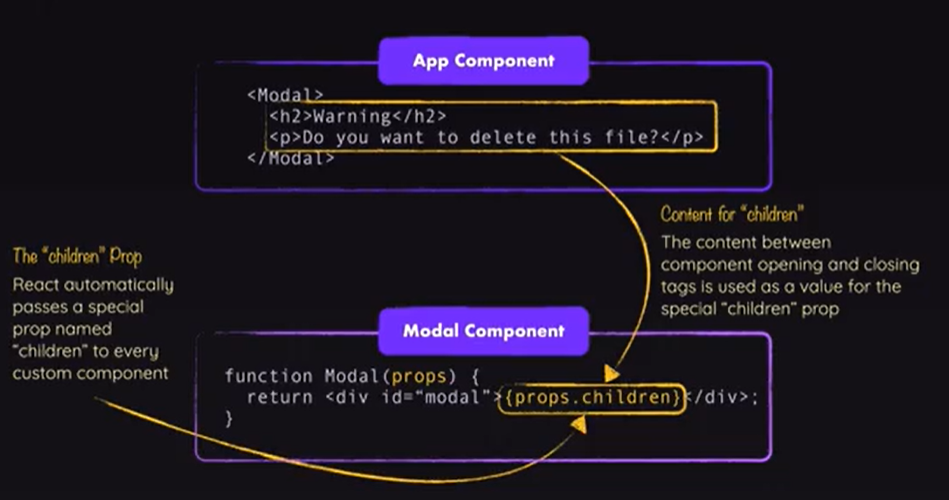
This can easily be achieved since JavaScript supports default values when using object destructuring:

export default function Button({ caption, type = "submit" }) {

// caption has no default value, type has a default value of "submit"

}

### **children PROP**



* The children prop is a special prop that allows you to pass components or elements as children to another component. It is used to create composite components and allows you to nest content within other components.
* In the below example, the ParentComponent has a children prop. When rendering the ParentComponent, we pass the <p> element as the children. The content of the children prop is then rendered inside the ParentComponent.

|  |
| --- |
| function ParentComponent({ children }) {  return (  <div>  <h1>Parent Component</h1>  {children} {/\* Render the children \*/}  </div>  );  }    function App() {  return (  <ParentComponent>  <p>This is the child component content.</p>  </ParentComponent>  );  } |

* The children prop can also be used to pass multiple components or elements as children:

|  |
| --- |
| function ParentComponent({ children }) {  return (  <div>  <h1>Parent Component</h1>  {children} {/\* Render the children \*/}  </div>  );  }    function App() {  return (  <ParentComponent>  <p>This is the first child component.</p>  <p>This is the second child component.</p>  </ParentComponent>  );  } |

## CREATING A DEMO REACT APP – BLOG POST APP

|  |  |  |
| --- | --- | --- |
|  | | Note: We will be using a third party for   * **CSS**: <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/semantic.min.css"> [TO BE ADDED IN index.html] * **AVATAR IMAGES**:   + <img src="https://source.unsplash.com/random" />   + OR BY USING FAKER NODE MODULE : <https://www.npmjs.com/package/faker>     - To install the node module : npm i faker * HTML : <https://semantic-ui.com/views/comment.html>   To create the blogpost component   1. We will one Parent Componet (Comments) . 2. The Parent componet has an **approval form component** 3. The Approval Component has child componenst “CommentDetails” . |
|  |  |
| **CONNECTING THE COMPONENTS**  **EXPORT**  export default CommentDetails;  **IMPORT**  import CommentDetails from './CommentDetails';  (This is the relative path of the JS file without having .js extension) |  |

### COMMUNICATION BETWEEN COMPONENTS – PARENT TO CHILD

|  |  |
| --- | --- |
|  | * React uses a “props” system to communicate between a Parent component to child/ nested components. |

#### CODE IMPLEMENTATION

|  |  |
| --- | --- |
| **Index.js**   * The App Component is a functional component which as Child Component “ApprovalCard” component * The ApprovalCard has in turn one more child component “CommentDetails “. | import React from 'react';  import ReactDOM from 'react-dom';  import faker from 'faker';  import CommentDetails from './CommentDetail';  import ApprovalCard from './ApprovalCard';  const App = () => {  return <div className="ui container comments">  <h3 className="ui dividing header">Comments</h3>  <div class="ui cards">  <ApprovalCard>  <CommentDetails name={faker.name.firstName()} />  </ApprovalCard>  <ApprovalCard>  <CommentDetails name={faker.name.firstName()} />  </ApprovalCard>  </div>  </div>;  };  ReactDOM.render(<App />, document.getElementById("root")); |
| **ApprovalCard.js**   * The ApprovalCard component can access its child component using “{props.children}” | import React from 'react';  const ApprovalCard = (props) => {  return <div className="card">  <div className="content">  {props.children}  </div>  <div className="extra content">  <div className="ui two buttons">  <div className="ui basic green button">Approve</div>  <div className="ui basic red button">Decline</div>  </div>  </div>  </div>;  };  export default ApprovalCard; |
| **CommentDetail.js**   * The value pass as an attribute from the parent can be accessed using “{props.*attributeName*}” | import React from 'react';  const CommentDetail = props => {  return <div className="comment">  <a href="/" className="avatar">  <img alt="avatar" src="https://source.unsplash.com/random" />  </a>  <div className="content">  <a className="author">{props.name}</a>  <div className="metadata">  <span className="date">Today at 5:42PM</span>  </div>  <div className="text">  How artistic!  </div>  <div className="actions">  <a className="reply">Reply</a>  </div>  </div>  </div>;  }  export default CommentDetail; |

## HANDLING EVENTS AND USER INPUTS

Application Name : Image Search

|  |  |
| --- | --- |
|  |  |

### BINDING EVENTS

|  |  |
| --- | --- |
| function ClickHandler() {  const handleClick = () => {  console.log("Click");  };  return (  <div>  <button onClick={handleClick}>Click</button>  </div>  );  } | INLINE THE HANDLER  function ClickHandler() {  return (  <div>  <button  onClick={() => {  console.log("Click");  }}> Click </button>  </div>  );  } |
| * Note – We do not put “()” while calling the event handler – otherwise it will be invoked when the component is loaded. | |

### CHILD TO PARENT COMMUNICATION

|  |
| --- |
| **PARENT**  function App() {  const handleClick = (selectedTab) => {  console.log(selectedTab);  };  return (  <div>  <TabButton onSelect={() => handleClick("components")}>Components</TabButton>  </div>  );  }  export default App; |
| **CHILD**  export default function TabButton({children,onSelect}) {  return (  <button onClick={onSelect}>{children}</button>  );  } |

## MANAGING STATES

### REACT HOOKS

React Hooks are a feature introduced in React 16.8 that allow you to use state and other React features in functional components, without the need for class components. Hooks provide a way to manage component state and lifecycle methods in a more concise and readable manner.

### BUILT IN HOOKS

There are several built-in hooks provided by React:

* **useState**: This hook allows you to add state to functional components. It returns an array with two elements: the current state value and a function to update that state.

**EXAMPLE**

|  |
| --- |
| import React, { useState } from 'react';  function Counter() {  const [count, setCount] = useState(0);  return (  <div>  <p>Count: {count}</p>  <button onClick={() => setCount(count + 1)}>Increment</button>  </div>  );  } |

**useEffect**:

* This hook allows you to perform side effects, such as fetching data, subscribing to events, or updating the DOM, in functional components.
* It is similar to the lifecycle methods componentDidMount, componentDidUpdate, and componentWillUnmount combined.

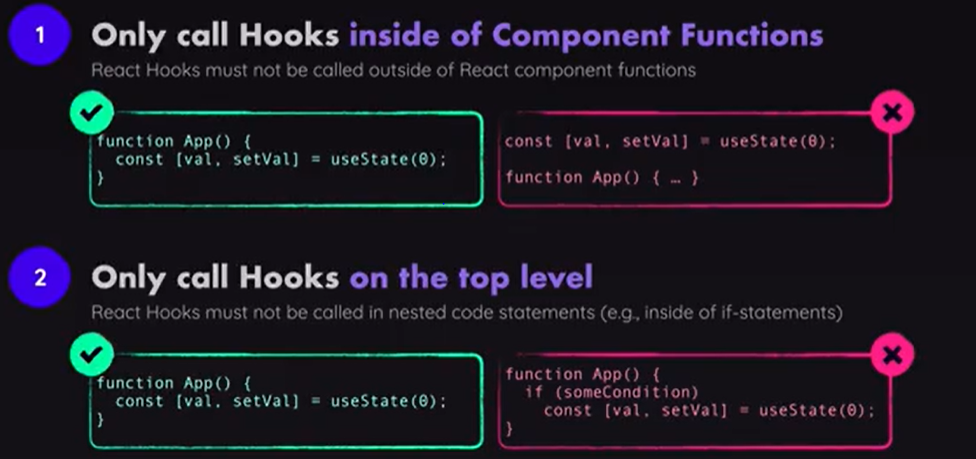
**EXAMPLE**

|  |
| --- |
| import React, { useEffect } from 'react';    function DataFetcher() {  useEffect(() => {  // Perform side effects here  console.log('Component mounted');    // Cleanup function  return () => {  console.log('Component unmounted');  };  }, []);    return <p>Data fetching component</p>;  } |

* **useContext**: This hook allows you to access the value of a React context within a functional component.

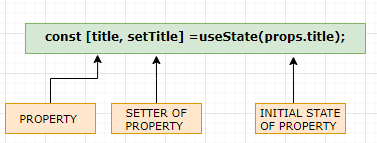
|  |
| --- |
| import React, { useContext } from 'react';    const MyContext = React.createContext();    function MyComponent() {  const value = useContext(MyContext);    return <p>Context value: {value}</p>;  } |

### RULES OF HOOKS



### STATES IN COMPONENTS

* Data that changes as user interacts with our application. When the data changes, react will update the content on the screen automatically



|  |  |
| --- | --- |
| import { useState } from 'react';  const ExpenseItem = (props) => {  **const [title, setTitle] =useState(props.title);**  const **changeTitle** =() =>{  setTitle("Updated");  }  return (  <h2>{title}</h2>  **<button onClick = {changeTitle}>Change Title</button>**  </div>  );  }  export default ExpenseItem; | * In the above example – “title” is a state variable which are getting managed using state variable. * The state of “title” property can be managed only by its setters. * ***Calling the setter cause the entire component to re-render and the updated state value.*** |

EXAMPLE 2: The primary focus of the below example is

* **LIST BUILDING IN REACT**
* **IMPORTING IMAGES**

|  |  |
| --- | --- |
| App.js | AnimalShow.js |
| import { useState } from "react";  import AnimalShow from "./AnimalShow";  function getRandomAnimal() {  const animalsList = ["bird", "cat", "cow", "dog", "gator", "horse"];  return animalsList[Math.floor(Math.random() \* animalsList.length)];  }  function App() {  const [animals, setAnimals] = useState([]);  function handleClick() {  const animal = getRandomAnimal();  setAnimals([...animals, animal]);  }  function renderAnimal() {  return animals.map((animal, index) => {  return <**AnimalShow** type={animal} key={index} />;  });  }  return (  <div>  <button onClick={handleClick}>Add Animals</button>  <p>{renderAnimal}</p>  </div>  );  }  export default App; | **import cow from "./images/cow.svg";**  **import cat from "./images/cat.svg";**  **import dog from "./images/dog.svg";**  **import gator from "./images/gator.svg";**  **import horse from "./images/horse.svg";**  **import heart from "./images/heart.svg";**  const svgIcons = {  cow,  cat,  dog,  gator,  horse,  };  function AnimalShow({ type, key }) {  return (  <div>  <img src={svgIcons[type]} alt="animal"/>  <img src={heart} alt="heart"/>  </div>  );  }  export default AnimalShow; |

#### STATE SYSTEM ELEMENT

|  |  |
| --- | --- |
| ADDING ELEMENT | const [colors, setColors] = useState([]);  const updatedColors = [... colors,newColor]  setColor(updatedColors) |
| ADDING ELEMENT ON SPECIFIC LOCATION | const [colors, setColors] = useState([]);  const addColorAtIndex = (newColor, index) => {  const updatedColors = [ ... colors.slice(0, index),  newColor, ... colors.slice(index)  ]  setColors(updatedColors);  }; |
| REMOVING ELEMENT WITH PARTICULAR VALUE |  |
| REMOVING ELEMENT WITH PARTICULAR INDEX | const [colors, setColors] = useState(['red', 'green', 'blue']);  const removeColorAtIndex = (indexToRemove) => {  const updatedColors = colors.filter((color, index)=>{  return index != indexToRemove  });  setColors(updatedColors)  }; |
| MODIFYING ELEMENT WITH PARTICULAR INDEX |  |
|  |  |

### CHILD TO PARENT COMMUNICATION

|  |  |
| --- | --- |
| **PARENT** | **CHILD** |
| function App() {  const handleClick = (term)=>{  // Some code  }  return (  <div className="App">  <SearchBar onSearch={handleClick}/>  </div>  );  }  export default App; | function SearchBar({onSearch}) {  const [term, setTerm] = useState("");  const handleOnSubmit = (event) => {  event.preventDefault();  onSearch(term)  };  const handleOnChange = (event) => {  setTerm(event.target.value);  };  return (  <form onSubmit={handleOnSubmit}>  <input value={term} onChange={handleOnChange} />  </form>  );  }  export default SearchBar; |

## CONDITIONS

VARIATION 1

|  |
| --- |
| **{!selectedTab && <p>Select a tab to see the example code</p>}**  **{selectedTab && <div id="tab-content">**  <h3>{EXAMPLES[selectedTab].title}</h3>  <p>{EXAMPLES[selectedTab].description}</p>  <pre>  <code>{EXAMPLES[selectedTab].code}</code>  </pre>  </div>} |

VARIATION 2(TERNIARY OPERARTOR)

|  |
| --- |
| {!selectedTab ? (  <p>Select a tab to see the example code</p>  ) : (  <div id="tab-content">  <h3>{EXAMPLES[selectedTab].title}</h3>  <p>{EXAMPLES[selectedTab].description}</p>  <pre>  <code>{EXAMPLES[selectedTab].code}</code>  </pre>  </div>  )} |

## STATE UPDATE

### ADDING ELEMENT

|  |  |
| --- | --- |
| Adding Elements to an Array: We can add elements to the start of an array by using the spread syntax. | const [colors, setColors] = useState(['red', 'green']);  const addColor = (colorToAdd) => {  const updatedColors = [colorToAdd, ...colors];  setColors(updatedColors);  }; |
| Add elements to the **end** of an array by reversing the order of elements in updatedColors. | const [colors, setColors] = useState(['red', 'green']);  const addColor = (colorToAdd) => {  // Now 'colorToAdd' will be at the end  const updatedColors = [...colors, colorToAdd];  setColors(updatedColors);  }; |
| * Elements can be added at any index by using the slice method available on all arrays. * The slice method can be used to add elements at the start or end of an array as well. | const [colors, setColors] = useState(['red', 'green']);  const addColorAtIndex = (colorToAdd, index) => {  const updatedColors = [  ...colors.slice(0, index),  colorToAdd,  ...colors.slice(index),  ];  setColors(updatedColors);  }; |

### REMOVING ELEMENT

|  |  |
| --- | --- |
| * Elements can be removed from an array by using the filter method. * The filter method can remove elements by index. | const [colors, setColors] = useState(['red', 'green', 'blue']);  const removeColorAtIndex = (indexToRemove) => {  const updatedColors = colors.filter((color, index) => {  return index !== indexToRemove;  });  setColors(updatedColors);  }; |
| filter can also remove elements by value. | const [colors, setColors] = useState(['red', 'green', 'blue']);  const removeValue = (colorToRemove) => {  const updatedColors = colors.filter((color) => {  return color !== colorToRemove;  });  setColors(updatedColors);  }; |

### CHANGING ELEMENT

|  |  |
| --- | --- |
| Objects in an array can be modified by using the map function. | const [books, setBooks] = useState([  { id: 1, title: 'Sense and Sensibility' },  { id: 2, title: 'Oliver Twist' },  ]);  const changeTitleById = (id, newTitle) => {  const updatedBooks = books.map((book) => {  if (book.id === id) {  return { ...book, title: newTitle };  }  return book;  });  setBooks(updatedBooks);  }; |

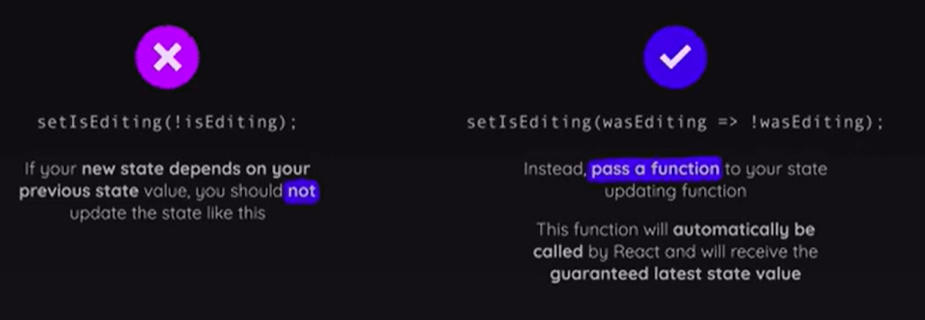
### CHANGING PROPERTIES

|  |  |
| --- | --- |
| **Changing Properties In Objects**  Properties in an object can be changed or added by using the spread syntax (the ...). | const [fruit, setFruit] = useState({  color: 'red',  name: 'apple',  });  const changeColor = (newColor) => {  const updatedFruit = {  ...fruit,  color: newColor,  };  setFruit(updatedFruit);  }; |

### REMOVING PROPERTIES

|  |  |
| --- | --- |
| Properties in an object can be removed by using destructuring. | const [fruit, setFruit] = useState({  color: 'red',  name: 'apple',  });  const removeColor = () => {  // `rest` is an object with all the properties  // of fruit except for `color`.  const { color, ...rest } = fruit;  setFruit(rest);  }; |

#### UPDATING STATE BASED ON OLD STATE



UP

## APPLYING DYNAMIC CSS CLASS

Example

 <button className={isSelected ? 'active': undefined} onClick={onSelect}>{children}</button>

## FORWARDING PROPS

* Forwarding props in React refers to passing props from a parent component to a child component without explicitly specifying each prop. This allows you to maintain the same prop names and values as they are received in the parent component.
* To forward props in React, you can use the spread operator (...) along with the props object. Here's an example:

|  |  |
| --- | --- |
| PARENT | CHILD |
| import Section from "./Section";  export default function Examples() {  return (  <Section id="examples">  ....  </Section>  );  } | export default function Section({title, children,...props}) {  return (  <section {...props}>  <h2>{title}</h2>  {children}  </section>  );  } |

## TWO WAY DATA BINDING

|  |
| --- |
| import { useState } from "react";  export default function Player({ initialName, symbol }) {  const [name, setName] = useState(initialName);  const handleChange = (event) => {  setName(event.target.value);  };  return (  <input type="text" value={name} onChange={handleChange} />  );  } |

## HTTP REQUEST IN REACT

In React paradigm the HTTP request can be accomplished using

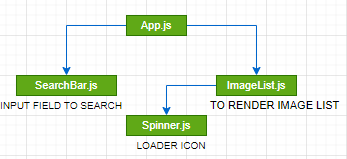
* XMLHTTPRequest or fetch API– Native JS Code
* Third party library called **Axios**

|  |  |
| --- | --- |
| INSTALLING AXIOS | npm install axios --save |

### APP OVERVIEW- SEARCH IMAGES

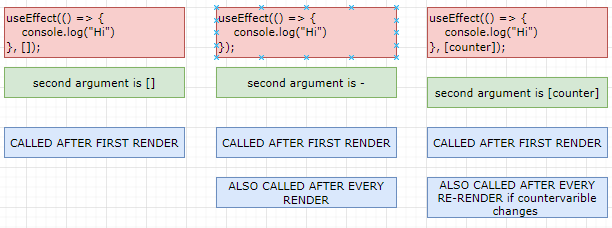
* GitHub repo : <https://github.com/avishekhsinhaRepo/React-Applications/tree/master/pics>

#### COMPONENT HIERARCHY



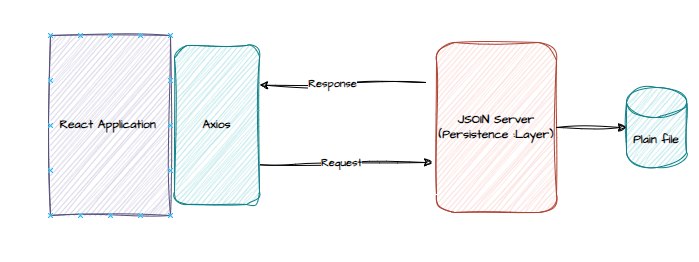
|  |  |
| --- | --- |
| **APP.JS**  import React from "react";  import SearchBar from "./SearchBar";  **import unsplash from "./api/unsplash";**  import ImageList from "./ImageList";  class App extends React.Component {  state = { images: [], isLoading: false };  onSearchEvent = async (term) => {  this.setState({ isLoading: true });  const response = **await unsplash**.get("/search/photos", {  params: {  query: term,  } });  this.setState({ images: response.data.results,isLoading: false });  };  render() {  return (  <div className="ui container" style={{ marginTop: "10px" }}>  <SearchBar onSearch={this.onSearchEvent} />  <ImageList imageList={this.state.images} isLoading={this.state.isLoading} />  </div>  )  }  }  export default App; | **SearchBar.js**  import React from 'react';  class SearchBar extends React.Component {  state= {term:''}  onFormSubmit=(event)=>{  event.preventDefault();  this.props.onSearch(this.state.term);  }  render() {  return (  <div className="ui segment">< form className="ui form" onSubmit={this.onFormSubmit}>  <div className="field">  <label>Image Search</label>  <input type="text" name="term" placeholder="Search" value={this.state.term} onChange={(e) => {this.setState({term: e.target.value})}} />  </div>  </form></div>);  }  }  export default SearchBar; |
| **Spinner.js**  const Spinner = (props) => {  return (  <div>  <div className="ui active dimmer">  <div className="ui text loader">{props.message}</div>  </div>  <p></p>  </div>  );  };  export default Spinner; |
| **ImageList.js**  import Spinner from "./Spinner";  const ImageList = (props) => {  const searchedImagesList = props.imageList.map((image) => {  return (  <img key={image.id} src={image.urls.thumb} alt={image.alt\_description} />  );  });  if (!props.isLoading) {  return <div>{searchedImagesList}</div>;  } else {  return <Spinner message="Fetching Images For You..." />;  }  };  export default ImageList; |
| **unsplash.js**  **This is a custom http client – getting called in App.js** |
| import axios from "axios";  export default axios.create({  baseURL: 'https://api.unsplash.com/',  headers: {  Authorization: "Client-ID  6S0qY-SVrLYgCbys4NiqHBXaiY0bKwywf7Ydl6FcmbU"  }  }); |

#### useEffect HOOKS



## DATA PERSISITENCE

* For development purpose – we can use JSON Server as persistence layer.
* JSON-server is a popular npm package that allows us to quickly create a RESTful API with JSON data.
* It serves as a mock server, allowing us to simulate a backend without the need for a full-fledged server.



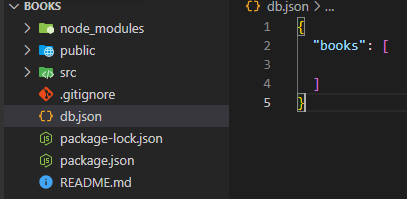
### SETTING UP JSON SERVER

|  |  |
| --- | --- |
| Command to install JSON Server | npm install json-server@0 |
| Create **db.json** file in the root directory of project (React App) |  |
| Set up the npm command in package.json & Run the command |  |

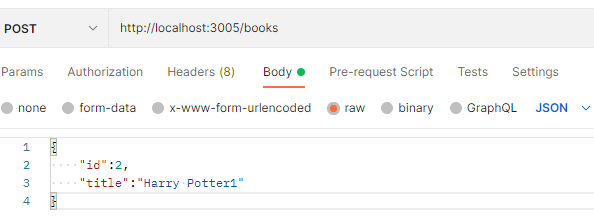
* In our case – the JSON server will store a list of books inside of a database and expose it as an API. React application is will work with this API by making requests for list of books or requests to edit,delete or create books
* **NOTE:** The API server is work with its local database and then eventually send a response back. The JSON-Server stores all of its information inside of a plain file(db.json)

### WORKING WITH JSON-SERVER

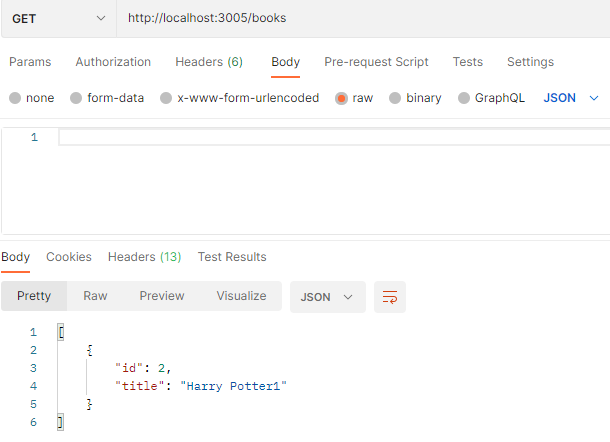
1. ADD ENTRY IN db.json



1. TO ADD DATA : POST CALL



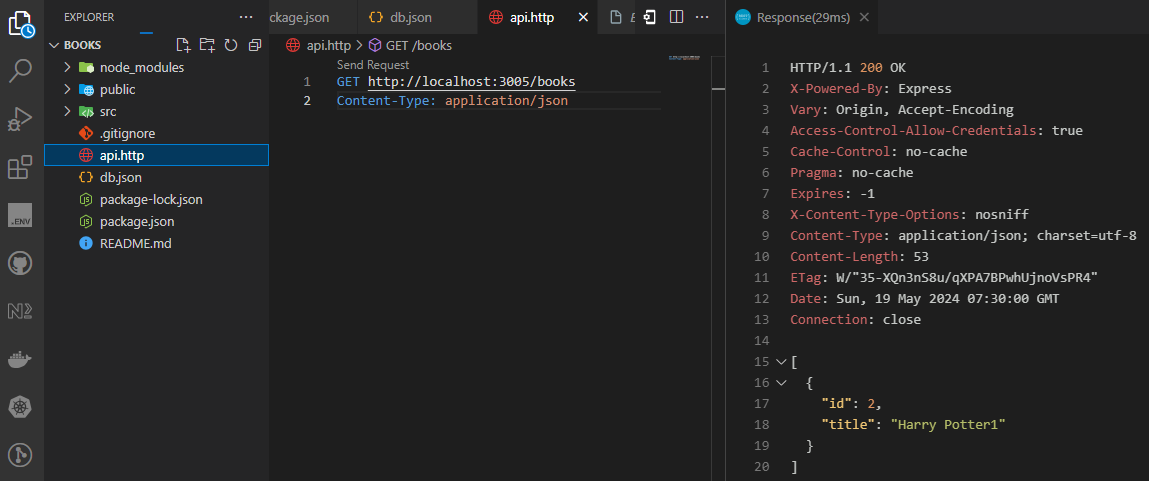
1. TO FETCH DATA



### USING REST CLIENT

|  |  |
| --- | --- |
| 1. Install the REST Client |  |

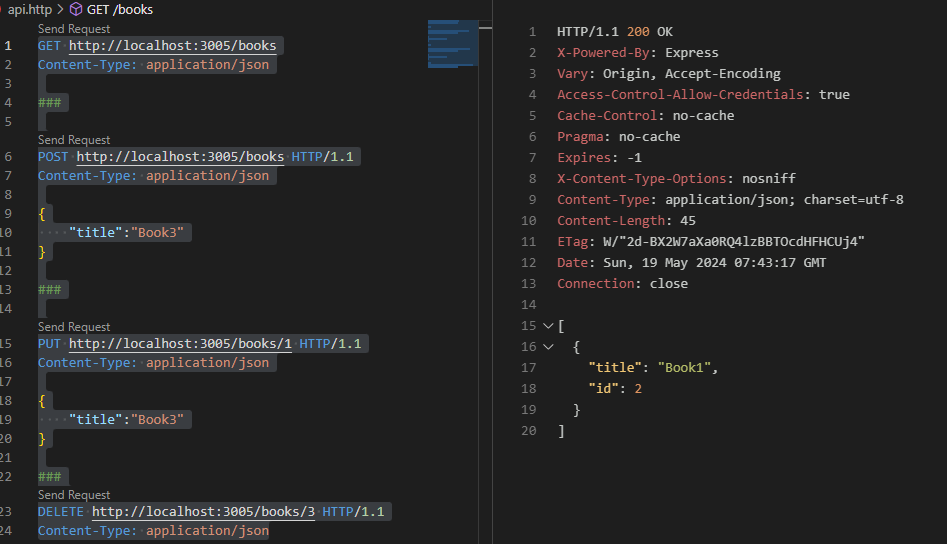
1. Create “**api.http**” file in the root folder of the project



Note : “Id” are auto-generated by JSON Server

**api.http**

|  |
| --- |
| GET http://localhost:3005/books  Content-Type: application/json  ###  POST http://localhost:3005/books HTTP/1.1  Content-Type: application/json  {      "title":"Book3"  }  ###  PUT http://localhost:3005/books/1 HTTP/1.1  Content-Type: application/json  {      "title":"Book3"  }  ###  DELETE http://localhost:3005/books/3 HTTP/1.1  Content-Type: application/json |



### UPDATING THE APP TO INTERACT WITH JSONSERVER

* Install axios
* Update the function

|  |  |
| --- | --- |
| **OLD WAY** | **UPDATED** |
| const createBook = async(title) => {  const updatedBooks = [  ...books,  {  id: Math.round(Math.random() \* 9999),  title,  },  ];  const updatedBooks = [  ...books,  response.data  ]  setBooks(updatedBooks);  }; | const createBook = async(title) => {  **const response = await axios.post("http://localhost:3005/books",{**  **title**  **})**  const updatedBooks = [  ...books,  response.data  ]  setBooks(updatedBooks);  }; |

### useEffect REACT HOOK

* The useEffect hook is a built-in hook in React that allows us to perform **side effects** in functional components. Side effects can include things **like fetching data from an API, subscribing to events, or manually manipulating the DOM**.
* To use the useEffect hook - import it from the react package: **import React, { useEffect } from 'react';**
* The useEffect hook takes two arguments:
  + Callback function : The function is where we put the code that we want to run when the component mounts or updates
  + The dependency array: The dependency array is an optional argument that allows us to control when the effect function is called. If the dependency array is empty ([]), the effect function will only run once when the component mounts.

|  |  |
| --- | --- |
| function MyComponent() {  const [count, setCount] = useState(0);    useEffect(() => {  console.log('Count changed:', count);  }, [count]);    return (  <div>  <p>Count: {count}</p>  <button onClick={() => setCount(count + 1)}>Increment</button>  </div>  );  } | * In this example**, the effect function will be called every time the count state variable changes**. * Remember that side effects like API requests or event listeners should be cleaned up when the component unmounts to avoid memory leaks. * We can return a cleanup function from the effect function to do this. The cleanup function will be called when the component unmounts or when the dependency array changes. |

#### VARIATIONS OF USE EFFECT

|  |  |
| --- | --- |
| **The arrow function will be called when the component is first rendered to the screen and then it will be never called again**, which of the following would you write? | useEffect(()=>{  console.log(counter)  }, **[]**); |
| **The arrow function will be called** when the component is **first rendered to the screen and whenever it re-renders** | useEffect(()=>{  console.log(counter)  }); |
| Imagine we are creating a component that has a state variable called counter. **The arrow function will be called**   * When the component is first rendered to the screen * When the component is re-rendered, but onl if counter changed | useEffect(()=>{  console.log(counter)  }, **[counter]**); |

## COMMUNICATION WITH CONTEXT SYSTEM

* Context provides a way to share data between components (without explicitly passing props through every level of the component tree) even if the component is not in child hierarchy.
* It allows us to create a global state or provide certain values to a subtree of components.

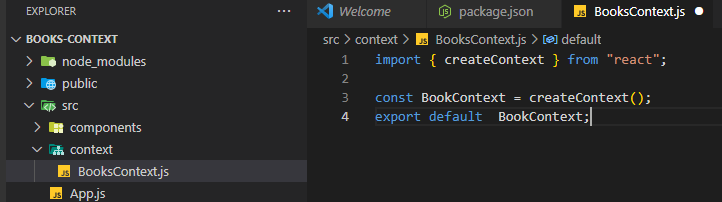
### SETTING UP THE CONTEXT

1. Step 1: Create a context
2. Step 2: Specify the data to be shared. The value we pass in there can be any type of value. It can be a simple number, string, array of strings, an object with different functions and nested data on it.
3. Step 3: Consume the data in the component.

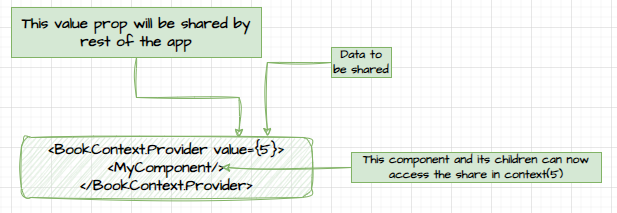
### SIMPLE CONTEXT EXAMPLE

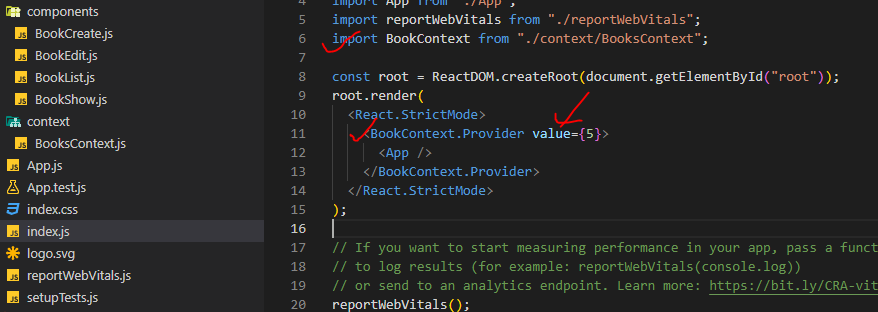
* In this simple example – we are going to create a context and share a data (some random integer), which will be consumed by other components.

STEP 1: CREATE A CONTEXT OBJECT: Create the context object in a separate file



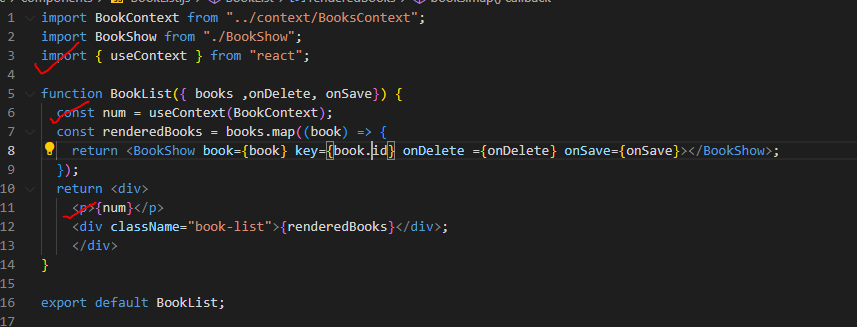
STEP 2: Data to be shared.





**The app component and all its children have access to that value prop.**

STEP3: CONSUMING THE SHARED/ CONTEXT DATA



### HOW TO UPDATE THE VALUE IN CONTEXT?

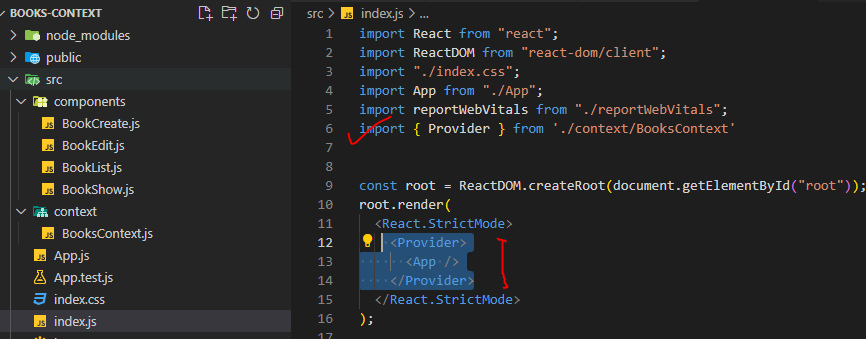
* The downside to above application is that the number five is static and fixed. To update the value in the context

|  |  |
| --- | --- |
|  | * Count will be a state which will cause the dependent component re-render when its value changes * “**incrementCounter**” is the function which will be called to update the count state variable * Both “count” and “incrementCounter” is passed as a value prop as an object |

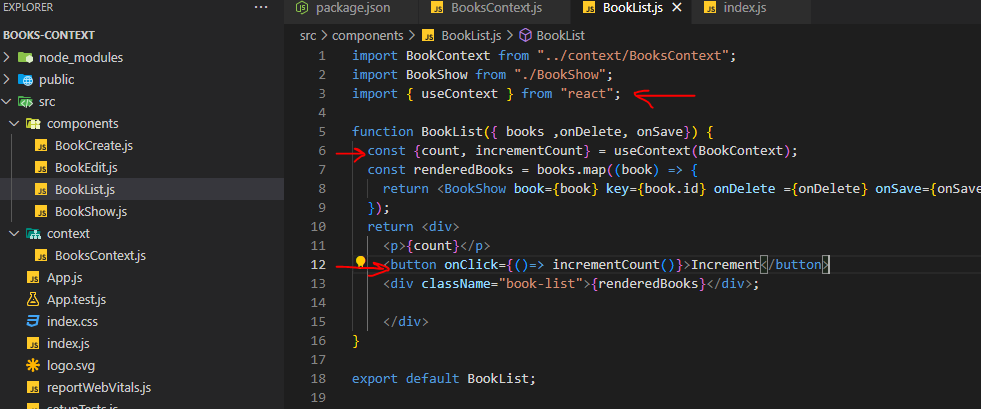
1. CREATE CONTEXT

|  |
| --- |
| import { createContext, useState } from "react";  const BookContext = createContext();  function Provider({ children }) {    const [count, setCount] = useState(5);    const valueToShare = {      count,      incrementCount: () => {        setCount(count + 1);      },    };    return <BookContext.Provider value={valueToShare}>      {children}    </BookContext.Provider>  }  export default BookContext;  export {Provider}; |

1. DATA TO BE SHARED : The App component will have access to the context variables/ values



1. CONSUMING THE CONTEXT



### EXAMPLE

* Github URL : [React-Applications/books-context at master · avishekhsinhaRepo/React-Applications (github.com)](https://github.com/avishekhsinhaRepo/React-Applications/tree/master/books-context)

## useReducer

* The useReducer hook in React is a built-in hook that allows us to manage complex state logic in functional components.
* It is an alternative to using the useState hook for managing state, especially when the state logic becomes more complex.
* useReducer is most useful.
  + **Whenever we have multiple pieces of state that are very closely related to each other.**
  + **Whenever we have some state where the future value of that state depends upon the current state.**

In the below example – where we have used “useState” hook. In the below example

1. The count state variable is dependent on valueToAdd state variable
2. The count state variable is dependent on its previous state

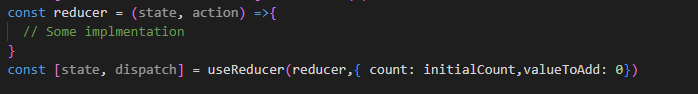
The implies that its perfect use case to “useReducer”

|  |
| --- |
| import { useState } from "react";  import Button from "../components/Button";  import Panel from "../components/Panel";  function CounterPage({ initialCount }) {  const [count, setCount] = useState(initialCount);  const [valueToAdd, setValueToAdd] = useState(0);  **const increment = () => {**  **setCount(count + 1);**  **};**  **const decrement = () => {**  **setCount(count - 1);**  **};**  const handleClick = (event) => {  event.preventDefault();  setCount(count + parseInt(valueToAdd));  setValueToAdd(0);  };  const handleOnChange = (event) => {  setValueToAdd(event.target.value || 0);  };  return (  <Panel className="m-3">  <h1 className="text-lg">Count is {count}</h1>  <div className="flex flex-row">  <Button onClick={increment}>Increment</Button>  <Button onClick={decrement}>Decrement</Button>  </div>  <form>  <label>Add a lot!</label>  <input type="number" className="p-1 m-3 bg-gray-50 border border-gray-300" onChange={handleOnChange} value={valueToAdd || ''} />  <Button onClick={handleClick}>Add it!</Button>  </form>  </Panel>  );  }  export default CounterPage; |

### CONCEPT



* useReducer is an alternative to useState.They are both hooks that are meant to produce state. They're both used to change state. And whenever that state changes, the component is going to automatically re-render.
* Whenever we make use of the useState hook, We call useState multiple times for each individual piece of state we want to declare. But When we make use of useReducer, We just one single state variable called “state”
* The state variable is an object that can have many different properties inside of it.
* if we want to access the state variable in a component, we would refer to **state.count**.



### RULES OF REDUCER FUNCTION

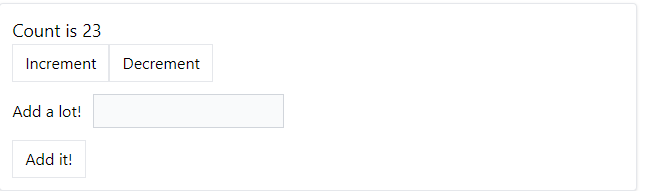
* That dispatch function triggers a state update. Hence, anytime we need to update our state- we need to call the dispatch function which in turn call the reducer function.
* When the function reducer is executed,
  + The first argument (state) will be the current state of the component.
  + The value of that second argument(action) will the value we passed into dispatch function.
  + Note: When we call dispatch, we can pass in no arguments or just 1 argument (If we pass in more than one argument, all the extras are completely ignored).
* **Now, what ever we return from the reducer function become the current state of the component**
* **If we return nothing from the reducer – the state will be undefined.**
* **We cannot use “async/ Await”, promises or outside variable in reducer. It should have just “action” & “state”**

### ACTION OBJECT

* The value passed from dispatch function will passed to action parameter in the reducer.
* Hence – among multiple dispatch events we can identify the respective events “type”(string) property

|  |
| --- |
| function CounterPage({ initialCount }) {  const reducer = (state, action) => {  const { type, payload } = action;  switch (type) {  case 'increment':  return {  ...state,  count: payload,  };  default:  return state;  }  };  const [state, dispatch] = useReducer(reducer, { count: initialCount, valueToAdd: 0 });  const increment = () => {  dispatch({  type: 'increment',  payload: state.count + 1,  });  };  return (  <Panel className="m-3">  <h1 className="text-lg">Count is {state.count}</h1>  <div className="flex flex-row">  <Button onClick={increment}>Increment</Button>  </div>  </Panel>  );  }  export default CounterPage; |

### EXAMPLE “useReducer”



|  |
| --- |
| import { useReducer } from "react";  import Button from "../components/Button";  import Panel from "../components/Panel";  function CounterPage({ initialCount }) {    const INCREMENT\_COUNT = "increment";    const DECREMENT\_COUNT = "decrement";    const CHANGE\_VALUE\_TO\_ADD = "change-value-to-add";    const ADD\_VALUE = "add-value";    const reducer = (state, action) => {      const { type, payload } = action;      switch (type) {        case INCREMENT\_COUNT:        case DECREMENT\_COUNT:          return {            ...state,            count: payload,          };        case CHANGE\_VALUE\_TO\_ADD:          return {            ...state,            valueToAdd: payload,          };        case ADD\_VALUE:          return {            ...state,            count: payload,            valueToAdd: 0,          };        default:          return state;      }    };    const [state, dispatch] = useReducer(reducer, { count: initialCount, valueToAdd: 0 });    const increment = () => {      dispatch({        type: INCREMENT\_COUNT,        payload: state.count + 1,      });    };    const decrement = () => {      dispatch({        type: DECREMENT\_COUNT,        payload: state.count - 1,      });    };    const handleClick = (event) => {      event.preventDefault();      dispatch({        type: ADD\_VALUE,        payload: state.count + parseInt(state.valueToAdd),      });    };    const handleOnChange = (event) => {      dispatch({        type: CHANGE\_VALUE\_TO\_ADD,        payload: event.target.value,      });    };    return (      <Panel className="m-3">        <h1 className="text-lg">Count is {state.count}</h1>        <div className="flex flex-row">          <Button onClick={increment}>Increment</Button>          <Button onClick={decrement}>Decrement</Button>        </div>        <form>          <label>Add a lot!</label>          <input type="number" className="p-1 m-3 bg-gray-50 border border-gray-300" onChange={handleOnChange} value={state.valueToAdd || ""} />          <Button onClick={handleClick}>Add it!</Button>        </form>      </Panel>    );  }  export default CounterPage; |

## IMMER

### IMMER IN ACTION

|  |  |
| --- | --- |
| INSTALL IMMER | npm install immer |
| IMPORT IMMER | import { produce } from "immer"; |
| REDUCER | const [state, dispatch] = useReducer(**produce**(reducer), { count: initialCount, valueToAdd: 0 }); |

|  |  |
| --- | --- |
| REDUCER WITHOUT IMMER | REDUCER WITH IMMER |
|  |  |

# CREATING REUSABLE COMPONENTS

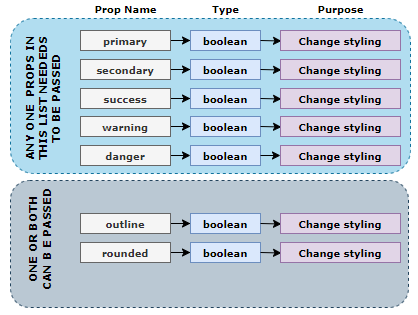
## REUSABLE BUTTON COMPONENT

|  |  |
| --- | --- |
| A group of buttons with text  Description automatically generated |  |

|  |  |
| --- | --- |
|  | * All the props will passed as Boolean value * The combination of props will uniquely determine the button type(as shown) * Example   **<Button danger outline rounded> Danger</Button>** |

### VALIDATION

* To set up the button component, we have 2 types of props
  + Out of the props - primary, secondary, success, warning and danger anyone can be passed as Boolean props
  + “outline” and “rounded” – both or either can be passed as Boolean props

--+

* Hence for such type of usecase we need validation of props, which can be achieved by a npm package - **prop-types**(<https://www.npmjs.com/package/prop-types> ).
* It validated whether props have been provided with correct “type”. If the validation fails – it will issue a warning (without breaking the app)
* We can implement a custom validation method(validator function) too for props validation

|  |  |
| --- | --- |
| **INSTALLING PROP TYPES** | npm i prop-types |
| **USAGE** | * Example :   + “title” 🡪 has to be a string and required. Validation has are nested   + “showImage” 🡪 has to a Boolean and optional |

#### CUSTOM FUNCTION – PROPS VALIDATION

**SIMPLE EXAMPLE**

|  |  |
| --- | --- |
|  | PASSING PROPS  <Button outline primary>Click me!</Button>  **OUTPUT** |

### STYLING THE BUTTON COMPONENT – TAILWINDCSS

* Tailwind CSS is a highly customizable, utility-first CSS framework that allows you to rapidly build user interfaces. It provides a set of pre-defined utility classes that you can directly apply to HTML elements to style them

|  |  |
| --- | --- |
| **SETTING UP TAILWIND CSS** | <https://tailwindcss.com/docs/guides/create-react-app> |

HOW TAILWIND IS DIFFERENT FROM OTHER CSS FRAMEWORKS

* UTILITY-FIRST APPROACH:
  1. Tailwind CSS follows a utility-first approach, which means it provides a large set of pre-defined utility classes that we can directly apply to HTML elements.
  2. Instead of writing custom CSS rules, we compose styles by combining these utility classes. This approach allows for rapid development and easy customization.
* HIGHLY CUSTOMIZABLE:
  1. Tailwind CSS is designed to be highly customizable.
  2. It provides an extensive configuration file where we can customize the default utility classes, add new classes, define color palettes, adjust breakpoints, etc.
* NO OPINIONATED STYLES:
  1. Unlike other CSS frameworks that come with pre-designed components and styles, Tailwind CSS focuses solely on providing utility classes.
  2. It does not impose any opinionated styles, giving complete control over the visual design.

EXAMPLE – TO MAKE THE BUTTON RED

Step 1 : Navigate to <https://tailwindcss.com/docs/>

Step 2: Press Ctrl+ K to search for CSS class

A blue and white stripe

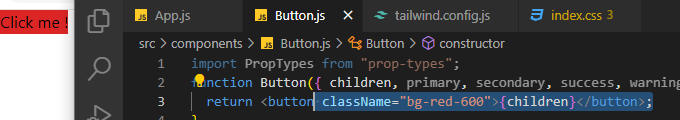
Description automatically generated

Step 3: Select the CSS class

A screenshot of a computer

Description automatically generated

Step 4: Apply the class to the HTML



|  |  |
| --- | --- |
|  | * By cascading multiple classes we can achieve the desired UI design |

#### CREATING VARIATIONS

|  |  |
| --- | --- |
|  | * In order to create different variations – the one few classes will change (in green box) and other remain as it is * For such use cases we need a utility which can dynamically create the list of classes * We can achieve it using a npm package “classnames” - <https://www.npmjs.com/package/classnames> |

##### Classnames PACKAGE

|  |  |
| --- | --- |
| INSTALL classname | npm install classnames |

USAGE

|  |  |  |
| --- | --- | --- |
|  | * It has a classNames method – the argument passed to it get combines and clear a list of string separated by spaces. * The resulting string can further be used on HTML components | |
|  | * The undefined values are ignored | |
|  | | * Here only those class names (key) will be appended whose value is truthy * For example here it will be “bg-blue-500” as primary is “truthy” |

###### BUTTON VARIATION SET UP

|  |  |
| --- | --- |
| import classnames from "classnames";  function Button({ children, primary, secondary, success, warning, danger, outline, rounded }) {    const buttonClass = classnames("px-3 py-1.5 border", {      "border-blue-500 bg-blue-500 text-white": primary,      "border-gray-900 bg-gray-900 text-white": secondary,      "border-green-500 bg-green-500 text-white": success,      "border-yellow-400 bg-yellow-400 text-white": warning,      "border-red-500 bg-red-500 text-white": danger,      "rounded-full": rounded,      "bg-white": outline,      "text-blue-500": outline && primary,      "text-gray-500": outline && secondary,      "text-green-500": outline && success,      "text-yellow-500": outline && warning,      "text-red-500": outline && danger,    });    return <button className={buttonClass}>{children}</button>;  }  export default Button;   * **Note: The classname added later will override the previously added classes . for example “bg-blue-500” in primary will be overridden by “bg-white” if <button className={primary outline}>Click Me</button>** | |
|  | **App.js**  import "./App.css";  import Button from "./components/Button";  function App() {    return (      <div>        <div>  **<Button success rounded outline>**  **Click me!!**  **</Button>**        </div>        <div>  **<Button danger outline>**  **Buy Now!**  **</Button>**        </div>        <div>  **<Button warning>See Deal!</Button>**        </div>        <div>  **<Button secondary outline>**  **Hide Ads!**  **</Button>**        </div>        <div>  **<Button primary rounded>**  **Something!**  **</Button>**        </div>      </div>    );  }  export default App; |

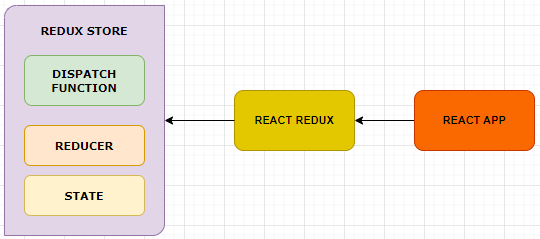
#### EVENT HANDER FOR BUTTON COMPONENT

|  |  |
| --- | --- |
| App.js | Button.js |
| import "./App.css";  import Button from "./components/Button";  function App() {  const handleOnClick = () => {  console.log("Button clicked!");  };  return (  <div>  <Button success rounded outline onClick={handleOnClick}>  Click me!!  </Button>  </div>  );  }  export default App; | import classnames from "classnames";  function Button({ children, primary, secondary, success, warning, danger, outline, rounded**, ...rest** }) {  const buttonClass = classnames("px-3 py-1.5 border", {  ...  });  return <button **{...rest}** className={buttonClass}>{children}</button>;  }  export default Button; |

# REDUX

* Redux is a library for managing the state which uses the same technique of useReducer Hook

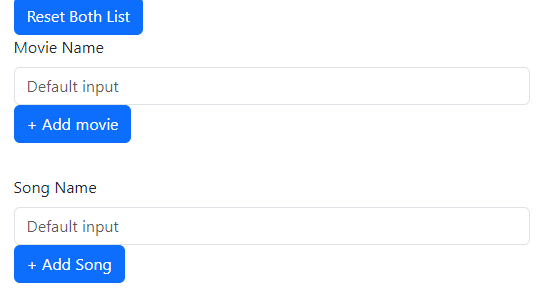
|  |  |
| --- | --- |
| INSTALLING REDUX TOOLKIT | npm install @reduxjs/toolkit |



* When we start making use of Redux, we are going to create a separate object called the Redux Store.
* This Redux store lives separately from our React application.(outside of our hierarchy of components). Anytime a component needs to get access to state, it’s will reach out to the Redux store and access the state that we have defined inside there.
* To interact with the store, we make use of called React Redux(This library is 100% optional).
* Redux uses the same strategy of useReducer, but slightly different.
  + useReducer - We make one single reducer function to manage all our states.

|  |  |
| --- | --- |
|  | * In Redux - We make use of multiple different reducer functions and each different reducer function is usually going to be in charge of managing a different part of our state. |
|  | STATE   * One other way in which Redux is a little bit different - state objects can * several different properties inside them. * For example, state object has some *users property,a videos property, and a messages property.* Each of these different properties are going to be created by a different reducer. |

## APP OVERVIEW – USING REDUX

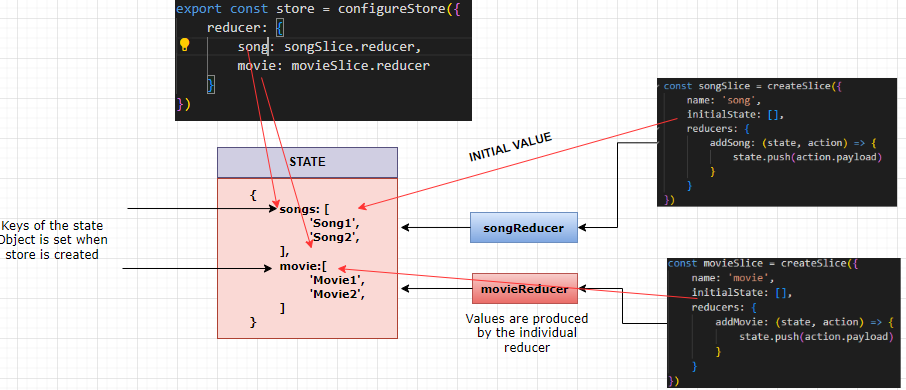


Application will create separate list of movies and songs

|  |  |
| --- | --- |
| STRUCTURE OF STATE  {  movies: [‘movie1’],  songs: [‘song1’],  } | * The structure of state object movies & songs property which are an array of strings. * On the list will be managed by a different reducers (each for movie list and song list ) * These reducer will change the list(movies/ songs) to add or remove movies and songs * **We will dispatch an action object to change our state**. * Example : user clicks on handleSongAdd 🡪 dispatch an action 🡪 tell the reducers of songs property in a state object to update songList. * The new state is going to be all kind of joined together into one big state object. |
|  | * We will create the store in a separate file |
| STORE | |
| **import { configureStore , createSlice } from '@reduxjs/toolkit';**  **const songSlice = createSlice({**  **name: 'song',**  **initialState: [],**  **reducers: {**  **addSong: (state, action) => {**  **state.push(action.payload)**  **}**  **}**  **})**  **const movieSlice = createSlice({**  **name: 'movie',**  **initialState: [],**  **reducers: {**  **addMovie: (state, action) => {**  **state.push(action.payload)**  **}**  **}**  **})**  **export const { addSong } = songSlice.actions;**  **export const { addMovie } = movieSlice.actions;**  **export const store = configureStore({**  **reducer: {**  **song: songSlice.reducer,**  **movie: movieSlice.reducer**  **}**  **})** | |

## UNDERSTANDING STORE

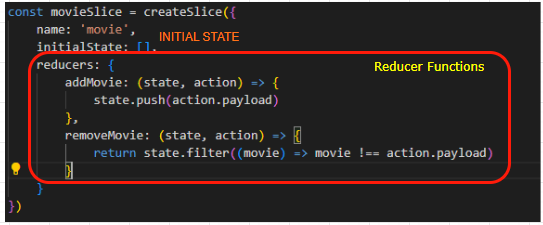
* In Redux, the store is a central container that holds the state of the application.
* It is responsible for managing the state and handling actions dispatched by the application.
* Usually, the application does not interact with redux store directly. The ‘React-Redux’ library will handle store for us.



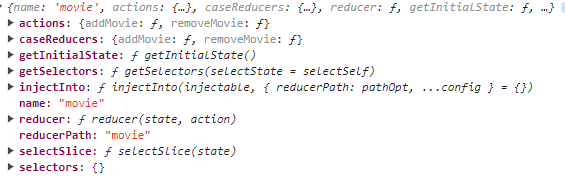
## UNDERSTANDING SLICES

The slices has following responsibilities

1. Define the initial state of a property in the state (object)
2. Combines ‘mini-reducers’ into big reducer
3. Creates a set of ‘action-creator’ function.



* Whenever we define a slice- we define set of mini-reducer functions on reducers property.
* In the above example - We feed it in to **createSlice** and we get back songsSlice.



* songsSlice iss an object, which has a property called reducer. That reducer property is like a big mega reducer which is combination of all of the individual reducers. This combined reducer is what actually gets loaded into the Redux store

HOW THE SPECIFIC REDUCER FUNCTION IS CALLED?

* The specific function(reducer) can be called using a specific pattern. For example to call **addSong**, the pattern will be

“<sliceName>/<reducer\_fn\_name>” i.e. “***song/addSong***”. They are action types .We don’t have to use specifically use these action types to call the reducer function. The tools which help us to create the action types automatically are called “action creators”.

## UNDERSTANDING ACTION CREATORS



* Whenever we create a slice using the **createSlice** function, It creates new object called action creators which has function exact same name as of reducer function. They are functions that are created for us automatically.
* The function in the Action creators return the action object (with type & payload property)
* Note : The action creator object can be accessed using “action” property using slice object

**songsSlice.actions**

|  |  |
| --- | --- |
| Accessing a specific action creator | songSlice.actions.addSong |
| Passing the Payload | songSlice.actions.addSong(“**Movie!!!!**”) |

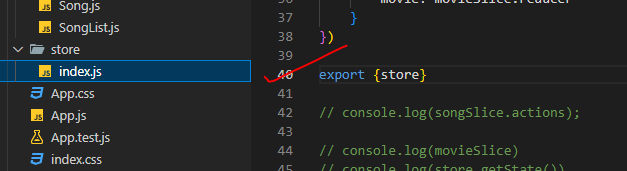
## CONNECTING REACT TO REDUX

* To access the store from the react app we make use of library called “React redux” library
* **The key part of the library is something called the Provider (like what we had in useContext). We will pass in our Redux store to it. Then through the context system, the store and specifically the dispatch function and the state within it are going to be made available to all the different components inside of our application.**

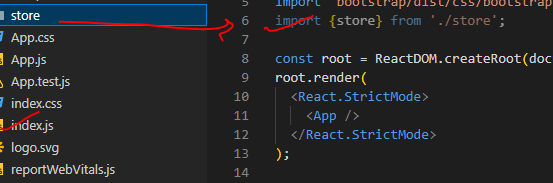
### STEPS TO CONNECT REDUX TO REACT

1. **Step 1: Export the store from the whatever file it is created.**
2. **Step 2: Import the store into the root “index.js” file.**
3. **Step 3: Import Provider from “react-redux” library.**
4. **Step 4: Wrap the App component with the provider & pass the store to the provider.**

**STEP 1: EXPORT THE STORE**: In the below example the store been exported as “named” export

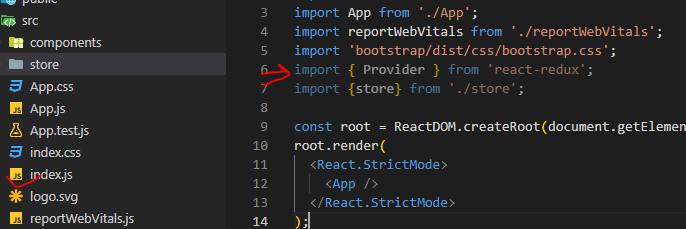


**STEP 2: IMPORT THE STORE**

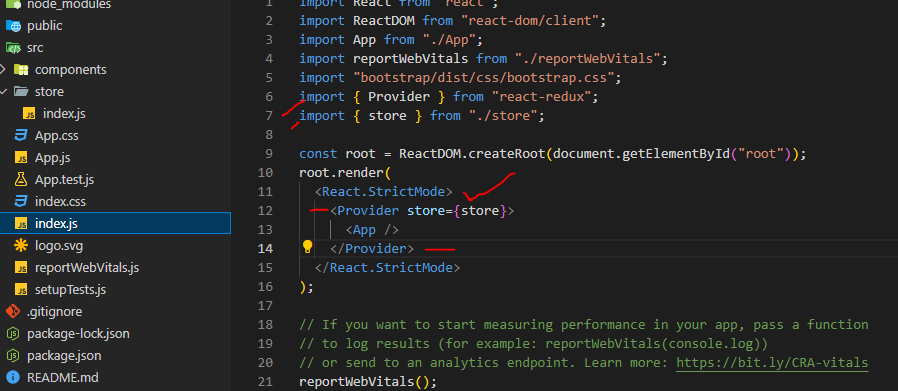


**STEP 3: IMPORT PROVIDER FROM “REACT-REDUX” LIBRARY.**

|  |  |
| --- | --- |
| **INSTALL react-redux library** | **npm install react-redux** |



**STEP 4: WRAP THE APP COMPONENT WITH THE PROVIDER & PASS THE STORE TO THE PROVIDER.**



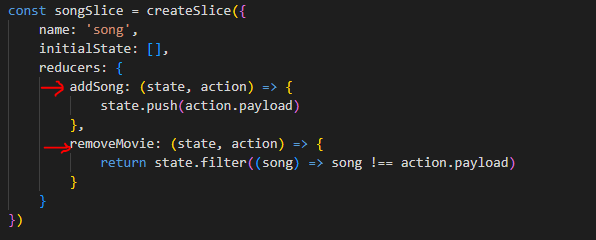
## ACCESSING STORE IN COMPONENT

* To access state inside of our store and make use of it from inside of our components, start to change our state in reaction to users doing stuff inside of our app.
* For example - whenever user clicks on add song to playlist, we update our state.

### STEPS

1. **Step 1: Add a reducer to one of the slices that changes state.**
2. **Step 2: Export the action creator that the slice automatically creates.**
3. **Step 3: Find the component that we want to dispatch**
4. **Step 4: import the action creator function and “useDispatch” from react-redux library**
5. **Step 5: Call the “useDispatch” hook to get access to dispatch function.**
6. **Step 6: When the user does something, call the action creator to get an action, then dispatch it.**

STEP 1: ADD A REDUCER TO ONE OF THE SLICES THAT CHANGES STATE



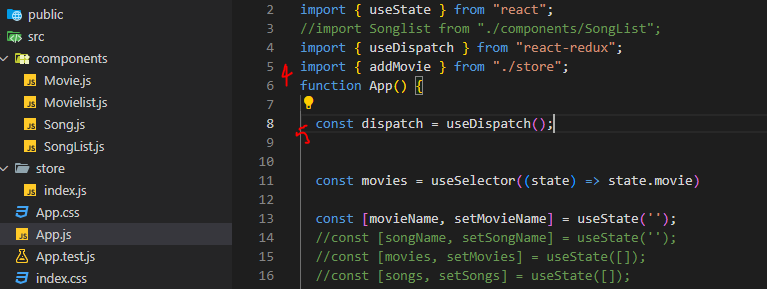
STEP 2: EXPORT THE ACTION CREATOR THAT THE SLICE AUTOMATICALLY CREATES

|  |
| --- |
| import { configureStore , createSlice } from '@reduxjs/toolkit';  const songSlice = createSlice({  name: 'song',  initialState: [],  reducers: {  addSong: (state, action) => {  state.push(action.payload)  },  removeMovie: (state, action) => {  return state.filter((song) => song !== action.payload)  }  }  })  const movieSlice = createSlice({  name: 'movie',  initialState: [],  reducers: {  addMovie: (state, action) => {  state.push(action.payload)  },  removeMovie: (state, action) => {  return state.filter((movie) => movie !== action.payload)  }  }  })  export const { addSong } = songSlice.actions;  export const { addMovie } = movieSlice.actions;  export const store = configureStore({  reducer: {  song: songSlice.reducer,  movie: movieSlice.reducer  }  }) |

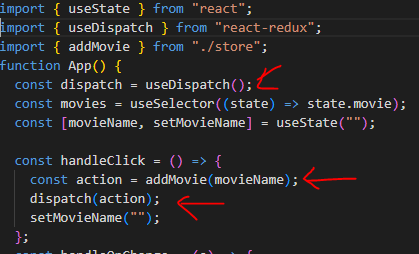
**STEP 3: FIND THE COMPONENT THAT WE WANT TO DISPATCH**

**STEP 4: IMPORT THE ACTION CREATOR FUNCTION AND “USEDISPATCH” FROM REACT-REDUX LIBRARY**

**STEP 5: CALL THE “USEDISPATCH” HOOK TO GET ACCESS TO DISPATCH FUNCTION.**



**STEP 6: WHEN THE USER DOES SOMETHING, CALL THE ACTION CREATOR TO GET AN ACTION, THEN DISPATCH IT.**

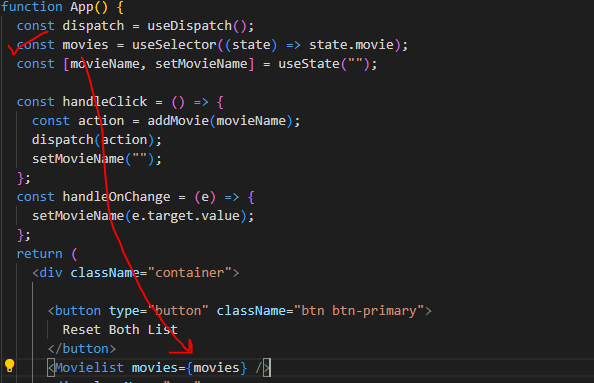


## UPDATING THE COMPONENT WITH STATE

Step 1: Import the “useSelector” hook from ‘react-redux’

Step 2: Call the hook, passing the selector function

Step 3: Use the state variable in the component.



### SELECTOR FUNCTION

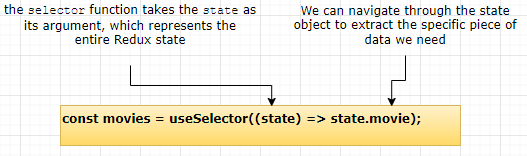
**const movies = useSelector((state) => state.movie);**

* The useSelector hook is a part of the Redux library in React, and it is used to extract data from the Redux store state. The selector function in useSelector is a callback function that takes the entire Redux state as its argument and returns the specific piece of data that you want to extract from the state.

**EXAMPLE**

|  |
| --- |
| import { useSelector } from 'react-redux';  const MyComponent = () => {  **const selectedData = useSelector(state => state.myReducer.myData);**  // Rest of your component logic...  return (  // JSX for your component...  );  } |

* In the example above, the selector function takes the state as its argument, which represents the entire Redux state. Inside the selector function, we can navigate through the state object to extract the specific piece of data we need. In this case, state.myReducer.myData is the data we want to extract from the Redux store.



* The selectedData variable will be updated whenever the selected data in the Redux store changes, and your component will re-render with the updated value.

**EXAMPLE – UPDATING THE COMPONENT WITH STATE**

|  |
| --- |
| import Movielist from "./components/Movielist";  import { useDispatch , useSelector} from "react-redux";  import { addMovie } from "./store";  function App() {  const dispatch = useDispatch();  const movies = useSelector((state) => state.movie);  const [movieName, setMovieName] = useState("");  const handleClick = () => {  const action = addMovie(movieName);  dispatch(action);  setMovieName("");  };  const handleOnChange = (e) => {  setMovieName(e.target.value);  };  return (  <div className="container">  <button type="button" className="btn btn-primary">  Reset Both List  </button>  <Movielist movies={movies} />  <div className="row">  <div className="mb-3">  <form>  <label htmlFor="movie-name" className="form-label">  Movie Name  </label>  <input className="form-control" type="text" placeholder="Default input" id="movie-name" value={movieName} onChange={handleOnChange} />  <button type="button" className="btn btn-primary" onClick={handleClick}> + Add movie </button>  </form>  </div>  </div>  </div>  );  }  export default App; |

## RESETING THE STATE