Table of Contents

1. Introduction

1.1 What is ReactJS?

React JS is a JavaScript library for creating user interfaces, making development of UI components easy and modular.

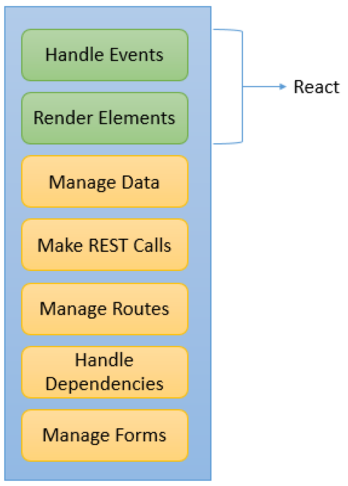
ReactJS or React is a client side UI library developed and maintained by Facebook. Currently Facebook uses React for many parts of its website.

React helps in creating interactive rich web application with multiple reusable components, handling data in an efficient manner and maintaining data flow in single direction which helps developers to create high performance web applications.

1.2 Library Vs Framework

A framework is full fledged solution for the different aspects of application development. A library on the other hand covers only a small part of the big picture and satisfies a specific requirement alone. Therefore we can use different libraries for different parts of the application.

React specifically caters to only two things: Rendering elements on the page efficiently and handling events. For everything else, we would have to write our own code or use other libraries. The below diagram shows the different needs of a typical complex front end application and which parts of it are covered by React.



1.3 Why ReactJS?

ReactJS is a JavaScript library. React helps us create SPA's which will dynamically load contents in a single HTML file, giving the user an illusion that the whole application is just a single page. One of the main reasons why React is more popular than other solutions and frameworks for SPA's is that React gives best performance in manipulating the DOM.

1.4 Features of ReactJS

- Lightweight and easy to use
- High performance DOM manipulation
- The data passed around is immutable and thus increases performance and reduces bugs
- Gives the flexibility to use other libraries
- Uses JavaScript instead of TypeScript
- It manages the state of the UI and updates only specific portions of the DOM

- **Component-based:** Components are the smallest unit in a React application. Anything that we want to render is rendered through components. Components help in maintainability and reusability.
- **Virtual DOM:** React uses virtual DOM concept for DOM manipulation which improves the performance of the application
- **Unidirectional data flow:** React's one-way data flow (also called one-way binding) keeps everything modular and fast and easy for debugging. The benefits of one-way data binding give you better control throughout the application. If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed. Flux is a pattern that helps to keep your data unidirectional. This makes the application more flexible that leads to increase efficiency.
- **JSX syntax:** React used JSX syntax which is similar to XML and HTML syntax which makes it easy for writing markup and binding events in components. JSX stands for JavaScript XML. It is a JavaScript syntax extension. Its an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but it is recommended to use in ReactJS.
- **SEO performance:** The SEO performance can be improved using the server-side rendering concept. We can develop isomorphic applications using React which increases the SEO performance.

1.5 What is Virtual DOM?

DOM is the tree representation of the various elements in an HTML page. When the browser receives an HTML page, it does the below steps:

- Parses the HTML content
- Creates the DOM structure
- The position and layout is calculated
- The DOM structure is painted on the browser

Whenever a DOM element changes, the browser re-calculates the position of the that element and its children elements and repaints them again. This is a costly process.

Imagine, if we have thousand DOM elements, then the re-positioning and re-painting operation done thousand times is a slow process. React simplifies it to a great extent using a Virtual DOM.

A virtual DOM is a in-memory representation of the actual DOM. The way virtual DOM works is:

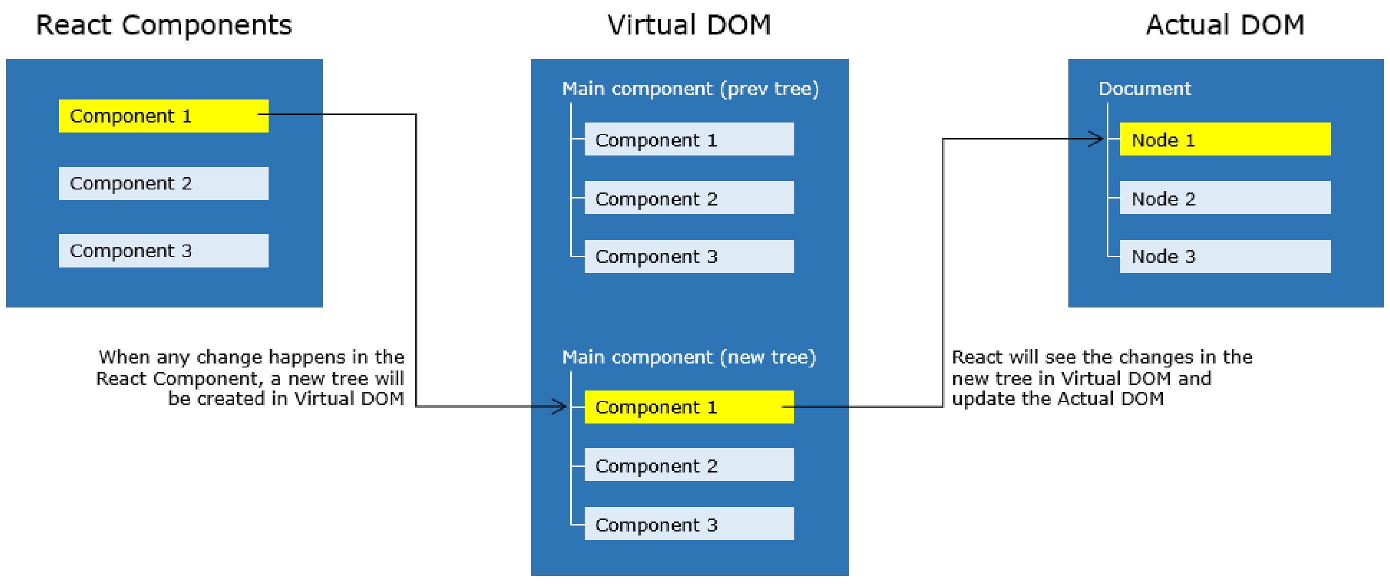
- The initial Virtual DOM is exact representation of the actual DOM
- When anything needs to be changed, it does not compare with the actual DOM. Instead it compares which parts of the Virtual DOM needs changes. This makes it faster.
- Instead of re-painting and re-positioning for every change, it creates a collection or batch of changes in the virtual DOM.
- It repaints only those areas in the DOM which have changed.

Virtual DOM is an abstraction of actual DOM, where components are the nodes. We can programmatically modify virtual DOM by updating components. These updates are internally handled by React and in turn, updated in actual DOM.

As we all know, DOM manipulation is expensive, because it requires traversal through entire DOM tree to find the element to be updated. If these updates are very frequent, this leads to a poor performance of an application.

Different frameworks handle above scenario in different ways like dirty checking, data-binding etc.

React, rather than updating DOM directly, builds an abstract version of it called Virtual DOM.


React Components
Component 1
Component 2
Component 3
When any change happens in the React Component, a new tree will be created in Virtual DOM
Virtual DOM
Main component (prev tree)
Component 1
Component 2
Component 3
Main component (new tree)
Component 1
Component 2
Component 3
Actual DOM
Document
Node 1
Node 2
Node 3
React will see the changes in the new tree in Virtual DOM and update the Actual DOM

1.6 What is "React Fiber"?

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM.

https://github.com/acdlite/react-fiber-architecture

1.7 Virtual DOM Vs Actual DOM

Real DOM	Virtual DOM
DOM is a language-neutral interface allowing programs and scripts to dynamically access and update multiple objects like content, structure, and style of a document.	Is a collection of modules designed to provide a declarative way to represent the DOM for an application.
The DOM represents the document as nodes and objects.	A virtual DOM object is a representation of a DOM object, like a lightweight copy.
It is an object-oriented representation of a web page, modified with a scripting language like JavaScript.	Virtual DOM is ideal for mobile-first applications.

1.8 Limitations of ReactJS

- React cannot be used for the entire front-end development
- One may have to depend on other libraries
- Choosing between multiple libraries can be difficult

1.9 React Vs Angular

Comparison Index	**Angular**	**React**
History	Angular is a TypeScript based JavaScript framework. It is written in TypeScript. Angular is developed and maintained by Google and known as a "*Superheroic JavaScript MVWFramework*". Angular is a complete rewrite of AngularJS. AngularJS was released in 2010 and it takes almost 6 years to release its second version Angular 2 (a complete rewrite). The latest version of Angular is Angular 8 now. Google AdWords which is one of the important project of Google uses Angular, so it is going to be big in upcoming years.	React is not a framework. It is a JavaScript library developed and maintained by Facebook and described as "a JavaScript library for building user interfaces. React was released in 2013 and after that it is being used at Facebook.
Architecture	Angular is a full MVC (Model,	React is a simple

	View, and Controller) framework. Angular is considered a framework because it provides strong facilities like how to structure your application. In Angular, you don't need to decide routing libraries. Most prominent features of Angular: ○ Provides templates, based on an extended version of HTML. ○ Provides XSS protection. ○ Provides Dependency injection. ○ Provides Ajax requests by @angular/HTTP. ○ @angular/router for Routing. ○ Component CSS encapsulation. ○ Utilities for unit-testing components. ○ @angular/forms for building forms.	JavaScript library (just the View) but it gives you much more freedom. React facilitates you to choose your own libraries. Most prominent features of React: ○ React uses JSX, an XML-like language built on top of JavaScript instead of classic templates. ○ XSS protection. ○ No dependency injection. ○ Fetch for Ajax requests. ○ Utilities for unit-testing components. React also provides some popular libraries to add functionalities: ○ React-router for routing. ○ Redux or MobX for state management. ○ Enzyme for additional testing utilities.
Used DOM	Angular uses regular DOM. The regular DOM updates the entire tree structure of HTML tags. It doesn't make difference in a simple real app but if you are dealing with large amount of data requests on the same page (and the HTML block is replaced for every page request), it affects the performance as well as the user's experience.	React uses virtual DOM which makes it amazing fast. It was said the most prominent feature of React when it was released. It updates only the specific part within a block of HTML codes. The virtual DOM looks only at the differences between the previous and current HTML and changes only that part which is required to be updated.
Used Templates	Angular uses enhanced HTML templates with Angular directives i.e. "ng-if" or "ng-for" etc. It is quite difficult	React uses UI templates and inline JavaScript logic together which was not done by any company

	because you have to learn its specific syntax.	before. This is called JSX. React uses component which contains both the markup AND logic in the same file. React also uses an XML-like language which facilitates developers to write markup directly in their JavaScript code. JSX is a big advantage for development, because you have everything in one place, and code completion and compile-time checks work better.
Data Binding	Angular provides two-way data binding. When you change the model state, then the UI element changes. In Angular, if you change the UI element, then the corresponding model state changes as well. Additionally, if you change the model state, then the UI element changes.	React provides one-way data binding. In React first the model state is updated, and then it renders the change in the UI element. When you change the UI element, the model state does not change.
TypeScript vs JavaScript	Angular is written in TypeScript, so you must be comfortable with TypeScript before using Angular.	React uses JavaScript which is a dynamically-typed language, so you don't have to define the variable's type. It makes it easy to use.
Scalability	Angular is easy to scale.	React is more scalable than Angular.
Speed	Angular is fast as compared to old technologies but React is faster than Angular.	React is faster than Angular.
Size	The size of Angular is large, so it takes longer load time and performance on mobile.	The size of React is smaller than Angular, so it is a little bit faster

1.10 Demo Create-React-App

Install node.js with version 10+ from the official site of Node.js

We can check the node version installed in the machine by running the following command which displays the node version as follows:

1. D:\> node -v
2. Install create-react-app by running the following command:

   D:\>npm install -g create-react-app
3. Once the installation is done, create a React app using the below command:

   D:\>create-react-app my-app

The description of the folder structure is:

Files	**Purpose**
node_modules	All the node module dependencies are created in this folder
Public	This folder contains the public static assets of the application
public/index.html	This is the first page that gets loaded when we run the application
Src	All application related files/folders are created in this folder
src/index.js	This is the entry point of the application
package.json	Contains the dependencies of the React application

To run the application, navigate to the folder **my-app** and run the command as shown below:

1. D:/>my-app>npm start

Observe the file src/index.js

```
ReactDOM.render(<App />, document.getElementById('root'));
```

ReactDOM.render is used to render an element or component to the virtual DOM.

- The first argument specifies what needs to be rendered
- The second argument specifies where to render.

The root element is present inside index.html

```
<body>
   <noscript>You need to enable JavaScript to run this app.</noscript>
   <div id="root"></div>
</body>
```

We can render elements also using the render method as shown below

```
ReactDOM.render(<h1>Hello React!<h1/>, document.getElementById('root'));
```

2. JSX

2.1 Why JSX?

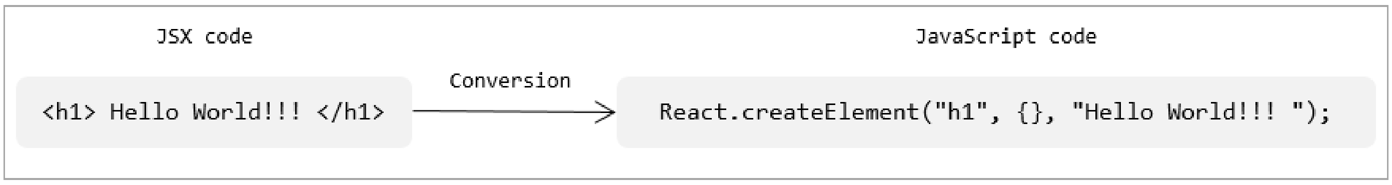
```
    class App extends React.Component{
       render(){
             return (
                   React.createElement('form', {},
                   React.createElement("h1", {}, "Login"),
                   React.createElement('input', {type: 'text',placeholder:'Name', value: ",}),
                   React.createElement('br', {}),React.createElement('br', {}),
                   React.createElement('input', {type: 'password', placeholder: 'password',
                                  value: ",}),
                   React.createElement('br', {}), React.createElement('br', {}),
                   React.createElement('button', {type: 'submit'}, "Login"))
                  )
            }
           };
     export default App;
```

We can observe from the above code, that we need to write more lines of JavaScript code to implement the Login component. The Code looks difficult to understand and hence productivity goes down.

JSX has been introduced in React to create elements that are very easy to read and write, which makes the component's code simple and understandable.

Let us see how to write the same above Login component using JSX in a easier way:

```
class App extends React.Component{
          render(){
                return (<form><h2>Login</h2>
                          <input type="text" placeholder="Name" /><br/><br/>
                          <input type="password" placeholder="password" /> <br/><br/>
                          <input type="submit" value="log" />
                       </form>);
                }
};
export default App;
```



```
<h1> Hello World!!! </h1>
```

```
React.createElement("h1", {}, "Hello World!!! ");
```

We saw that JSX is converted into React.createElement() JavaScript code.

Syntax:

```
React.createElement(arg1,arg2,arg3);
```

This method creates a HTML element and renders it.

- arg1 - this will be the name of the tag which gets rendered
- arg2 - this will be data which will be passed along with element rendering
- arg3 - this will be the children of the rendered element

When working with React components, we may need to render multiple React elements. For example, let us consider the below code where we need to render multiple React elements:

```
class App extends React.Component {
render(){
return <h3>ReactJS:</h3>
<img src="./image/react.PNG" width="120" height="120"/>
<p> React is a JavaScript library for creating User Interfaces.</p>

}}
```

The above code logs an error saying "Adjacent JSX elements must be wrapped in an enclosing tag". As per the JSX syntax, all the adjacent elements must be wrapped in an enclosing tag i.e. there should be only the outermost element. Hence the above code can be modified as follows:

```
class App extends React.Component {
render(){
return <div>
<h3>ReactJS:</h3>
<img src="./image/react.PNG" width="120" height="120"/>
<p> React is a JavaScript library for creating User Interfaces.</p>
</div>
}
}
```

By adhering to JSX syntax we use <div> for grouping the elements and this introduces an extra and unnecessary node into the DOM. As a solution to this, **React Fragments** are introduced which is a common pattern in React for a component to return multiple elements. **React Fragments** allows us to group a list of React elements without adding any extra node to the DOM.

```
class App extends React.Component {
render(){
return <React.Fragment>
<h3>ReactJS:</h3>
<img src="./image/react.PNG" width="120" height="120"/>
<p> React is a JavaScript library for creating User Interfaces.</p>
</React.Fragment>

}
}
```

2.2 Accessing a variable

```
const productId=128;
const element = <h1>Product ID : {productId}</h1>
```

2.3 Accessing an object

```
const product={
productId: 128,
price: 1200
}
const element = <p>Product ID:{product.productId}</p>
```

2.4 JSX with attributes

An attribute's value can be specified as an expression using curly braces as follows:

```
    const imgUrl="product1.PNG"
    const element = <img src={imgUrl} alt="This is a product image" />
```

2.5 Conditional Rendering

2.5.1 Using if-else:

```
class App extends React.Component {
    render() {
        let isLoggedIn = true
        if(isLoggedIn) {
            return <h2>Welcome Admin</h2>
         }
        else{
             return <h2>Please Login</h2>
        }
    }
```

The above code can also be written as shown below:

```
class App extends React.Component {
      render() {
             let element = null;
             let isLoggedIn = false
             if(isLoggedIn) {
                    element = <h2>Welcome Admin</h2>
              }
             else {
                     element = <h2>Please Login</h2>
             }
             return (<React.Fragment>
                    {element}
             </React.Fragment>)
      }
}
export default App;
```

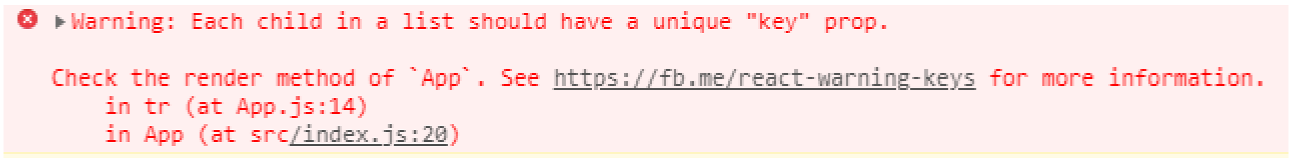
2.5.2 Using ternary operator:

```
class App extends React.Component {
      render() {
             let element = null;
             let isLoggedIn = false
             isLoggedIn ? element = <h2>Welcome Admin</h2> : element = <h2>Please
Login</h2>

             return (<React.Fragment>
                    {element}
             </React.Fragment>)
      }
}
```

2.5.3 Looping using map() method

```
class App extends React.Component {
      render() {
              var employees = [
                { empId: 1234, name: 'John', designation:'SE' },
                { empId: 4567, name: 'Tom', designation:'SSE'},
                { empId: 8910, name: 'Kevin',designation:'TA'}
              ]

             return (<React.Fragment>
                    <table style={{width:'60%'}} className='table'>
                        <thead className="thead-light">
                           <tr>
                               <th>EmpID</th>
                               <th>Name</th>
                               <th>Designation</th>
                           </tr>
                        </thead>
                        <tbody>
                            {employees.map(employee => {
                                 return (<tr>
                                        <td>{employee.empId}</td>
                                        <td>{employee.name}</td>
                                        <td>{employee.designation}</td>
                                 </tr>)
                           })
                           }
                        </tbody>
                    </table>
             </React.Fragment>)
 export default App;
```

The above code provides a warning in the console as shown below.



```
▸Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`. See https://fb.me/react-warning-keys for more information.
    in tr (at App.js:14)
    in App (at src/index.js:20)
```

To overcome the above error, we have to provide a unique key prop to each element in an array as shown:

```
<tbody>
   {employees.map(employee => {
      return (<tr key={employee.empId}>
         <td>{employee.empId}</td>
         <td>{employee.name}</td>
         <td>{employee.designation}</td>
       </tr>)
    })
    }
</tbody>
```

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

Because generally list items are huge in number. Another reason is that when you add or remove items from the list, it is best to render only that item on the DOM and others remain as they were.

This can only be possible when React knows what is added and where.

3. Components

Components are the basic building blocks of a React application, which encapsulates data, presentation logic and the business logic as a single unit. A react application will have one root component and other components will be nested in them.

The main advantages of splitting the application into multiple components are:

- Each component can be developed independent of other components
- Each component can be reused in other components as well
- We can update one component without affecting other, thus avoiding a complete page refresh when data changes

3.1 Class Components

To create a component we need to create a class which extends the **React.Component** class and implement the render(). The render() must return a JSX.

```
import React from "react";
class HelloComp extends React.Component {
  render() {
    return (
        <h1>Hello New Component</h1>
    );
  }
}
export default HelloComp;
```

Component Name

Component name should be in PascalCase which means first letter should always be capital.

Valid: class MyComp extends React.Component {}

Invalid: class myComp extends React.Component {}

If your Component name is not in PascalCase then your React application will throw an error because React consider tags starting with lowercase letter as HTML/DOM tags and tags starting with Uppercase letter as Component.

render()

method will render the component's elements. Should have a mandatory return, which returns a JSX.

If you don't implement render() method in any class Component you will get TypeError while when you run your react application. You must always implement render() method while creating any class Component. Error will be: TypeError: instance.render is not a function

return ()

One should always enclose return of Component with parentheses () because if after return our code start with next line, JavaScript will automatically put a semicolon after return and any code from next will not execute.

A React component class should follow the rules:

- For creating class methods, arrow function syntax must be used.
- If a constructor is being used, a super(); method call should be the first line of the constructor if this keyword is used in the constructor.

Observe the code below:

```
class HelloComp extends React.Component {
constructor() {
super();
this.name = "Jack";
}

//arrow functions should be used
// code might behave unpredictably if arrow not used!!
displayName = () => {
return (
<h1>Hello, {this.name}'s World</h1>
);
}

render() {
return (
<div>
{this.displayName()}
</div>
);
}
}
```

In the above code, the HelloComp has name as an instance variable. We are initializing this variable in the constructor. super() should be called the first inside the constructor.

We are using this.name in the class method displayName(), to access name.

The method displayName() uses arrow function syntax.

For rendering this component, we have placed the following code in index.js:

```
ReactDOM.render(<HelloComp></HelloComp>, document.getElementById('root'));
```

3.1.1 Props in Class Component

- Props allow us to pass data from one component to another component
- Props are immutable - a component cannot change its props however it is responsible for putting together
- Props can be accessed as this.props

In order to pass data to a class component as attributes, we need to do two things:

- Pass values as attributes wherever the component is being rendered.
- Update the component to accept the data through props.

Note: The component re-renders if props are updated from parent component.

The updated **index.js** will look like below:

```
ReactDOM.render(<HelloComp greeting="Morning" name="Jack"/>, document.getElementById('root'));
```

In the above code we are passing two parameters to the *HelloComp*: **greeting** and **name**.

To access these values in the *HelloComp*, we need to use props as shown below:

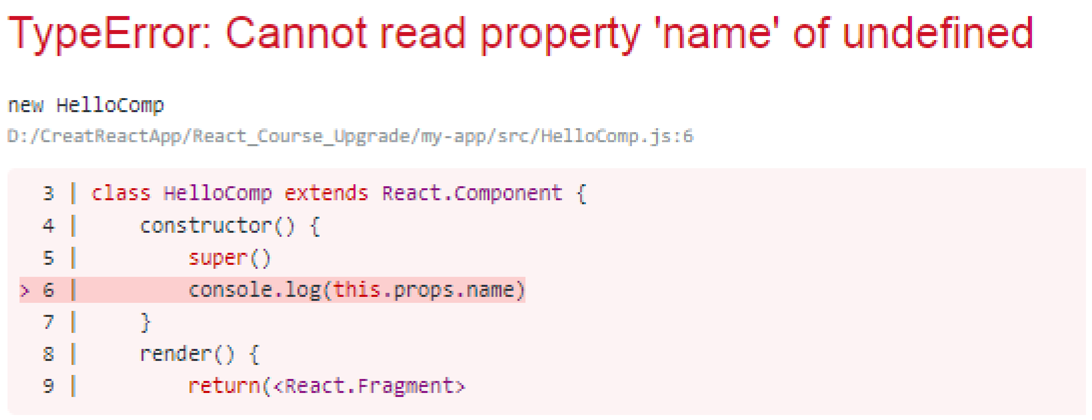
```
class HelloComp extends React.Component {
   render() {
      return (
         <h2>Hello {this.props.name}, Good {this.props.greeting}</h2>
      );
   }
}
```

The value of the props can be access by any method of the component using **this.props.** Here we are using **this.props** to access the *name* and *greeting* data passed as attributes.

If we want to access props within constructor as shown below:

```
class HelloComp extends React.Component {

constructor(){
super();
console.log(this.props.name, this.props.greeting)
}

render() {
return (
<h2>Hello {this.props.name}, Good {this.props.greeting}</h2>
);
}
}
```

We get below Error:



TypeError: Cannot read property 'name' of undefined

```
new HelloComp
D:/CreatReactApp/React_Course_Upgrade/my-app/src/HelloComp.js:6

class HelloComp extends React.Component {
    constructor() {
        super()
        console.log(this.props.name)
    }
    render() {
        return(<React.Fragment>
```

This is because *this.props* is undefined in the constructor, even though it is available in all the other methods of the component.

To avoid this, we need to make two changes:

- The constructor must take *props* as an argument
- The props must be passed as a parameter to the *super()* invocation.

The below constructor will not throw an error:

```
constructor(props){

super(props);

console.log(this.props.name, this.props.greeting)

}
```

Alternative Syntax:

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

Default Prop Values

You can define default values for your props by assigning to the special defaultProps property:

```
AppComp.defaultProps = {

    element1 : "Hello",

            element2: "default value",

}
```

In case there is no value passed to the prop and if it is being accessed within the component, then the default values will be considered. Here, we need to mention to which component, we need to set the default values for props.

PropTypes

React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special propTypes property:

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,

  // Anything that can be rendered: numbers, strings, elements or an array
  // (or fragment) containing these types.
  optionalNode: PropTypes.node,

  // A React element.
  optionalElement: PropTypes.element,

  // A React element type (ie. MyComponent).
  optionalElementType: PropTypes.elementType,

  // You can also declare that a prop is an instance of a class. This uses
  // JS's instanceof operator.
  optionalMessage: PropTypes.instanceOf(Message),

  // You can ensure that your prop is limited to specific values by treating
  // it as an enum.
  optionalEnum: PropTypes.oneOf(['News', 'Photos']),

  // An object that could be one of many types
  optionalUnion: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
    PropTypes.instanceOf(Message)
  ]),

  // An array of a certain type
  optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

  // An object with property values of a certain type
  optionalObjectOf: PropTypes.objectOf(PropTypes.number),

  // An object taking on a particular shape
  optionalObjectWithShape: PropTypes.shape({
    color: PropTypes.string,
    fontSize: PropTypes.number
  }),

  // An object with warnings on extra properties
  optionalObjectWithStrictShape: PropTypes.exact({
    name: PropTypes.string,
    quantity: PropTypes.number
  }),

  // You can chain any of the above with `isRequired` to make sure a warning
  // is shown if the prop isn't provided.
  requiredFunc: PropTypes.func.isRequired,

  // A required value of any data type
  requiredAny: PropTypes.any.isRequired,

  // You can also specify a custom validator. It should return an Error
  // object if the validation fails. Don't `console.warn` or throw, as this
  // won't work inside `oneOfType`.
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error(
        'Invalid prop `' + propName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  },

  // You can also supply a custom validator to `arrayOf` and `objectOf`.
  // It should return an Error object if the validation fails. The validator
  // will be called for each key in the array or object. The first two
  // arguments of the validator are the array or object itself, and the
  // current item's key.
  customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location,
propFullName) {
    if (!/matchme/.test(propValue[key])) {
      return new Error(
        'Invalid prop `' + propFullName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  })
};
```

3.1.1.1 Render Props

The Render Props is a technique in ReactJS for sharing code between React components using a prop whose value is a function. Child component takes render props as a function and calls it instead of implementing its own render logic. In brief, we pass a function from the parent component to the child component as a render props, and the child component calls that function instead of implementing its own logic.

```
import React from 'react'

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Render Props Example</h1>
        <SampleRenderProps />
      </div>
    )
  }
}

// Child component getting render props
class RenderPropsComponent extends React.Component {
  render() {
    return (
      <div>
        <h2>I am Child Component</h2>
        {this.props.render()}
      </div>
    )
  }
}

// Parent component sending render props to the child
class SampleRenderProps extends React.Component {
  render() {
    return (
      <RenderPropsComponent
        // Passing render props to the child component
        render={() => {
          return (
            <div>
              <h3>
               I am coming from render props
              </h3>
            </div>
          )
        }}
      />)}}
export default App;
```

3.1.1.2 Children Props

Some components don’t know their children ahead of time

We recommend that such components use the special children prop to pass children elements directly into their output:

Represents the content between the opening and the closing tags when invoking/rendering a component.

```
import React from 'react';
class App extends React.Component {
        render() {
                return ( <ul>
                                {this.props.children}
                        </ul>)
        }
};
export default App;
```

Render App component to the DOM within index.js as shown below:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App>
      <li>List element</li>
      <h3>Heading element</h3>
      <p>Paragraph element</p>
      <span>Span element</span>
</App>, document.getElementById('root'));
```

3.1.2 State in Class Component

The state is an instance of React Component Class can be defined as an object of a set of observable properties that control the behavior of the component.

In other words, the State of a component is an object that holds some information that may change over the lifetime of the component.

Using constructor, the state of a component is created and initialized using **this.state** as follows:

```
// With constructor
class WithConstructor {
  constructor() {
   this.state = {
     name: "Bhargav"
    }
  }
}
```

Another way of initializing state in React is to use the Class property. Once the class is instantiated in the memory all the properties of the class are created.

```
// Without constructor
class WithoutConstructor {

state = {
name: "Bhargav"
}
}
```

State of a component can be updated using ***setState()*** method.

- We can pass a new *object literal* to ***setState(*****{..., ..., ...}*****)***
- This new object literal should represent the updated state

```
Class HelloComp extends React.Component {
        constructor() {
        super();
        this.state = {
                counter: 0
        };
        }
        incrementCounter = () => {
        //this.state.counter += 1;  //trying to increment(or mutate) the state!
        this.setState({counter : this.state.counter + 1 }) //updating state using setState()
        console.log("Button Clicked," , this.state.counter, "times" );
         };
        render() {
        return (
        <div>
           <button onClick={this.incrementCounter}> Click </button>
           <p>{this.state.counter}</p>
        </div>
     );
        }
}
```

Whenever any state updates happens, *setState()* method causes re-rendering of a component to reflect the changes.

Important Note:

- If the attributes in the state object are existing, it will update them, else add the new attributes in the updated object literal.
- We can not re-assign the state without using *setState()*.

What if we want to *access the newly updated state*, and further update it?

Modify the ***incrementCounter*** method as shown below. Re-run the code and observe.

```
incrementCounter = () => {

     this.setState({counter : this.state.counter + 1 });

     this.setState({counter : this.state.counter + 1 }); //trying to use the updated state

     this.setState({counter : this.state.counter + 1 }); //trying to use the updated state again

     console.log("Button Clicked," , this.state.counter, "times" );

};
```

The code does not work as intended. Instead of displaying "Button Clicked 3 times" it displays "Button Clicked 1 times". This is because setState() updates **what is currently rendered.** Since 0 is what is currently rendered, calling setState() multiple times updates 0 only.

If we want to update the state **based on previous state**,

- the *setState()* method can take a *callback* with a parameter. This param holds the previous state of the component.
- The callback should return a new object literal having the updated state attributes.
- The final syntax will be: *setState(**(**previousState**) => {** return **{..., ..., ...} }**)*

Observe the updated *incrementCounter's* code below:

```
incrementCounter = () => {
        this.setState({counter : this.state.counter + 1 });
        this.setState((prevState) => {
            return {counter: prevState.counter + 1}
        });
        this.setState((prevState) => {
            return {counter: prevState.counter + 1}
        });
        console.log("Button Clicked," , this.state.counter);
};
```

Final note on setState()

- Do not call *setState()* inside *render()*. It causes an infinite loop.
- Do not call *setState()* in constructor. Use *this.state = {...}*, for initializing state for the first time.
- *setState()* should be called from events like button clicks or from life-cycle methods.

3.1.3 Props Vs State

Both state and props make up the data used by a component. A component converts this data into HTML presentation. Though both are data, they have certain similarities and differences as shown below:

State	Props
This is the data owned by the component	It is the data passed to the component as attributes
State is mutable in the component through *setState()*	Props are immutable in the component
It is a JS object	It is a JS object

3.1.4 Lifecyle Methods

Every component will have following phases in its Lifecycle:

- Mounting phase - when the component is mounted to DOM tree
- Updating phase - when component is being updated with new state or new props are being received
- Unmounting phase - destroying component from DOM tree
- Error Handling - Handling errors within React Component

Every phase in the lifecycle of a component has few methods which will be invoked during that phase of a component's lifecycle. We can override these methods to provide the desired functionality.

These methods can be used in the following cases:

- For making an Ajax call, to set timer and to integrate with other frameworks
- To avoid unnecessary re-rendering of a component
- To update the component, based on the props changes
- For clearing the values when component is unmounted

3.1.4.1 Mounting Phase:

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- constructor()
- static getDerivedStateFromProps()
- render()
- componentDidMount()

3.1.4.1.1 constructor()

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

- Initializing local state by assigning an object to this.state.
- Binding event handler methods to an instance.

You should not call setState() in the constructor(). Instead, if your component needs to use local state, assign the initial state to this.state directly in the constructor:

Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use componentDidMount() instead.

3.1.4.1.2 static getDerivedStateFromProps()

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

Since it is a static method, we cannot access 'this' keyword within this method. Hence we cannot call setState method.

This method exists for rare use cases where the state depends on changes in props over time.

This method is used to update the state of a component based on the changes of the props as shown below

```
class Employee extends React.Component {
  static getDerivedStateFromProps(props, state) {
    if (props.isLoggedIn !== state.isLoggedIn) {
      return {
        isLoggedIn: props.isLoggedIn,
      };
    }
    return null;
  }
}
```

3.1.4.1.3 render()

Every React component must have render() method. render() method should be a pure function which returns the same result every time it is called which means, it doesn't modify any component's state.

render() will not be invoked if shouldComponentUpdate() returns false.

3.1.4.1.4 componentDidMount()

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here.

Best place for setting timers and handling Ajax request.

Demo: The Timer will start as shown below, as soon as the component is mounted

Timer Component

```
import React from 'react';
class Timer extends React.Component {
        constructor() {
            super();
            this.state = {
                secondsElapsed: 0
            };
        }
        start = () => {
            this.setState({
                secondsElapsed: this.state.secondsElapsed + 1
            });
        }
        componentDidMount() {
            this.interval = setInterval(this.start, 1000);
        }
        render() {
            return ( <React.Fragment>
                    <h2> Seconds Elapsed: {this.state.secondsElapsed} </h2>
                </React.Fragment>);
            }
        }
export default Timer;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import Timer from './Timer';
ReactDOM.render(<Timer/>, document.getElementById('root'));
```

3.1.4.2 Updating Phase:

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

3.1.4.2.1 static getDerivedStateFromProps()

Same as 3.1.4.1.2

3.1.4.2.2 shouldComponentUpdate(nextProps, nextState)

Use shouldComponentUpdate() to let React know if a component's output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

shouldComponentUpdate() is invoked before rendering when new props or state are being received. Defaults to true.

If it returns false componentDidUpdate() method will not be invoked

In case we want to skip re-rendering, this method could be used by returning false, so this would skip re-rendering

This method only exists as a performance optimization.

3.1.4.2.3 render()

Same as 3.1.4.1.3

3.1.4.2.4 getSnapshotBeforeUpdate(prevProps,prevState)

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to componentDidUpdate().

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

This method is the last chance to get the prevState and prevProps values before the component is updated. The method can return values based on the prevState and prevProps values.

A snapshot value (or null) should be returned.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    if (prevState.chats.length < this.state.chats.length) {
      const layout = this.layout.current;
      const isAtBottomOfChatWindow =
        window.innerHeight + window.pageYOffset === layout.scrollHeight;
      return { isAtBottomOfChatWindow };
    }
    return null;
}
```

The above method checks whether the user is at the bottom of the chat window. If so, then the user should be scrolled to the bottom of the chat window, when new messages comes up in the chat window.

If user is not at the bottom of the chat window, then keep them there. Hence we return null

The snapshot returned from getSnapshotBeforeUpdate method will be passed as a third argument to the componentDidUpdate method as shown below

```
componentDidUpdate(prevProps, prevState, snapshot) {

  if (snapshot.isAtBottomOfChatWindow) {
    window.scrollTo({
      top: this.layout.current.scrollHeight
    });
  }
}
```

In the componentDidUpdate method, we check whether user was at the bottom on chat window using snapshot argument. If yes, we scroll the user to the bottom of the new messages that has come.

3.1.4.2.5 componentDidUpdate(prevProps,prevState,snapshot)

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

You may call setState() immediately in componentDidUpdate() but note that it must be wrapped in a condition or you’ll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance.

componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.

Demo: Component will be updated only if the value is >3

Timer Component

```
import React from 'react';
class Timer extends React.Component {
   constructor(props) {
   super(props);
   this.state = {
      count: 0
      };
   }
   start = () => {
      this.setState({
      count: this.state.count + 1
      });
   }
   componentDidMount() {
      this.interval = setInterval(this.start, 2000);
   }
   render() {
      return ( <React.Fragment>
               <Updates new = {this.state.count} />
            </React.Fragment>);
         }
      }
class Updates extends React.Component {
   shouldComponentUpdate(newProps, newState) {
      if (this.props.new <= 3) {
         console.log('shouldComponentUpdate:', newProps);
            return false;
         } else {
             return true;
         }
   }
   componentDidUpdate(prevProps, prevState, snapshot) {
         console.log('Previous value destroyed:', prevProps);
         console.log('Updated');
   }
   render() {
      return ( <React.Fragment>
               <h2> Seconds Elapsed: {this.props.new} </h2>
            </React.Fragment>);
         }
}
export default Timer;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import Timer from './Timer';
ReactDOM.render(<Timer/>, document.getElementById('root'));
```

3.1.4.3 Unmounting Phase:

This method is called when a component is being removed from the DOM.

3.1.4.3.1 componentWillUnmount()

componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

You should not call setState() in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

A component can be unmounted from DOM using ReactDOM.unmountComponentAtNode()

Demo:

Component will be unmounted from DOM after 15 seconds and observe how componentWillUnmount() is used for code cleanup

Timer component

```
import React from 'react';
class Timer extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         count: 0
      };
   }
   start = () => {
      this.setState({
      count: this.state.count + 1
      });
   }
   componentDidMount() {
      this.interval = setInterval(this.start, 2000);
   }
   componentWillUnmount() {
      console.log('Component WILL UNMOUNT!')
      clearInterval(this.interval);
   }
   render() {
      return ( <React.Fragment>
            <Updates new = {this.state.count} />
         </React.Fragment>);
      }
   }
   class Updates extends React.Component {
      shouldComponentUpdate(newProps, newState) {
         if (this.props.new <= 3) {
            console.log('shouldComponentUpdate:', newProps);
               return false;
            } else {
               return true;
         }
      }
      componentDidUpdate(prevProps, prevState, snapshot) {
         console.log('Previous value destroyed:', prevProps);
         console.log('Updated');
      }
      render() {
         return ( <React.Fragment>
                  <h2> Seconds Elapsed: {this.props.new} </h2>
               </React.Fragment>);
            }
}
export default Timer;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import Timer from './Timer';
ReactDOM.render(<Timer/>, document.getElementById('root'));
setTimeout(() => {
   ReactDOM.unmountComponentAtNode( document.getElementById('root') );
  }, 15000);
```

3.1.4.4 Error Handling:

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.
Error boundaries should be defined in order to catch errors occurring within the child components.
Whenever a component crashes because of a JavaScript error, the error will be logged and fallback UI is displayed.
Error boundaries are nothing but class components which defines either getDerivedStateFromError or componentDidCatch method.

3.1.4.4.1 static getDerivedStateFromError(error)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }}
```

getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted. For those use cases, use componentDidCatch() instead.

3.1.4.4.2 componentDidCatch(error,info)

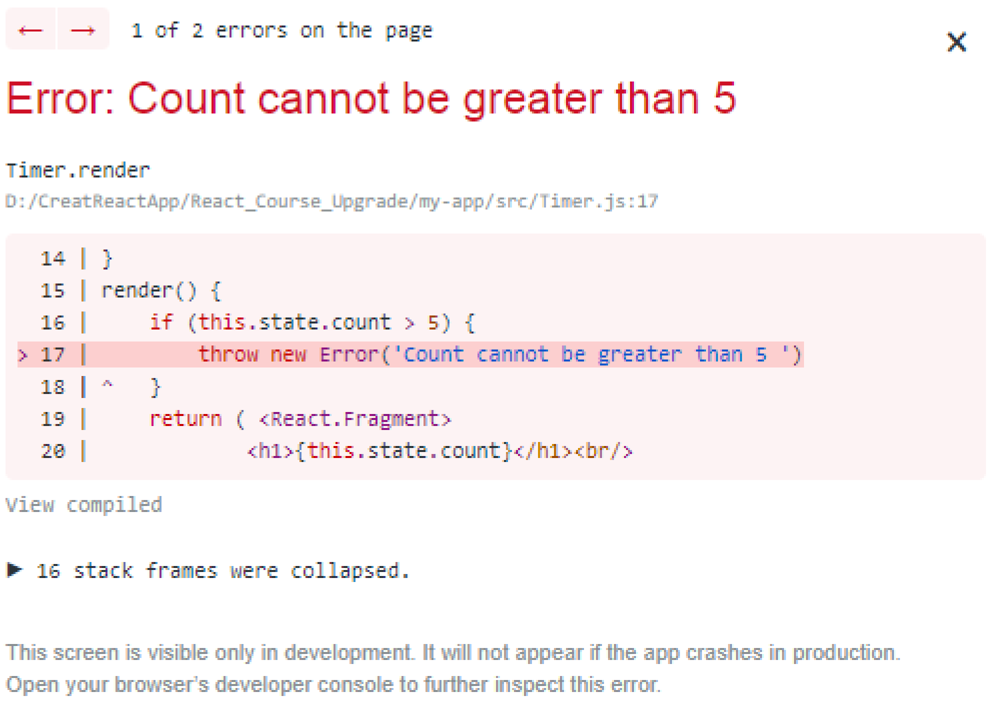
This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

- error - The error that was thrown.
- info - An object with a componentStack key containing information about which component threw the error.

componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors.

```
class Timer extends React.Component {
constructor(props) {
super(props);
this.state = {
count: 0
};
}
start = () => {
this.setState({
count: this.state.count + 1
});
}
render() {
if (this.state.count > 5) {
throw new Error('Count cannot be greater than 5 ')
}
return ( <React.Fragment>
<h1>{this.state.count}</h1><br/>
<button onClick={this.start}>Update</button>
</React.Fragment>);
}}
```

The above component throws an error as shown below when the value of count is greater than 5.



```
1 of 2 errors on the page
Error: Count cannot be greater than 5

Timer.render
D:/CreatReactApp/React_Course_Upgrade/my-app/src/Timer.js:17

  14 | }
  15 | render() {
  16 |     if (this.state.count > 5) {
> 17 |         throw new Error('Count cannot be greater than 5 ')
  18 | ^   }
  19 |     return ( <React.Fragment>
  20 |             <h1>{this.state.count}</h1><br/>
View compiled

▶ 16 stack frames were collapsed.

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.
```

Here we will not be able to get to know, the error occurred in which component.

In order to know the component tree, we need to define Error boundaries, as shown below

```
import React from 'react';
class ErrorHandler extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        error: false,
        errorInfo: null,
      };
    }

    componentDidCatch(error, info) {
      this.setState({
        error: error,
        info: info,
      });
    }

    render() {
      if(this.state.error) {
        return (
          <React.Fragment>
            <h5>Sorry. Counter value is greater than 5</h5>
            <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
            <br />
            {this.state.errorInfo}
          </details>
          </React.Fragment>
        );
      }
      return this.props.children;
    }
  }
export default ErrorHandler;
```

The ErrorHandler is just a class component which implements the componentDidCatch method.

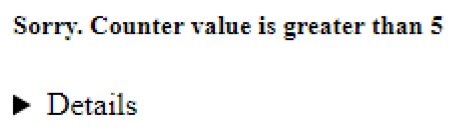
The componentDidCatch method receives the error and info as arguments.

error is the actual error message which tells the reason for the error and info contains stack trace of the error

Now wrap the Timer component within the ErrorHandler component as shown below

```
ReactDOM.render(<ErrorHandler>
<Timer/>
</ErrorHandler>,document.getElementById('root'));
```

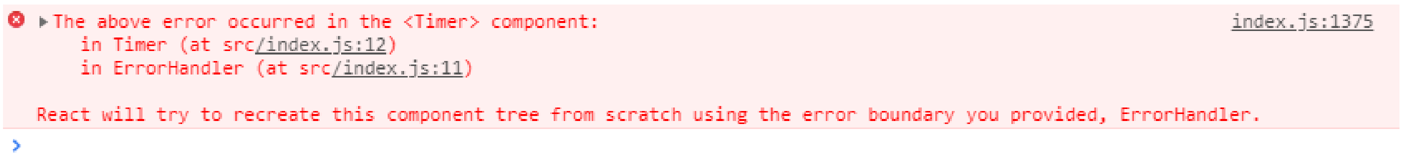
Now when the counter value is greater than 5, the error is displayed as shown below



Sorry. Counter value is greater than 5

▶ Details

The stack trace is also displayed in the console, where it logs the exact component where the error occurred.



```
▶The above error occurred in the <Timer> component:                                    index.js:1375
    in Timer (at src/index.js:12)
    in ErrorHandler (at src/index.js:11)

React will try to recreate this component tree from scratch using the error boundary you provided, ErrorHandler.
>
```

3.1.5 Context API

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context API helps in passing data through the component tree without the problem of passing props down to every component in the hierarchy.

Context helps in making the data available to all components throughout the component hierarchy, no matter how deeply nested the components are.

3.1.5.1 React.createContext

While using context API first we would create a context object.

```
const MyContext = React.createContext(defaultValue);
```

Creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.

The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This default value can be helpful for testing components in isolation without wrapping them.

Note: passing undefined as a Provider value does not cause consuming components to use defaultValue.

3.1.5.2 Context.Provider

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.

```
<MyContext.Provider value={/* some value */}>
```

The Provider component accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

All consumers that are descendants of a Provider will re-render whenever the Provider's value prop changes. The propagation from Provider to its descendant consumers (including .contextType and useContext) is not subject to the shouldComponentUpdate method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as Object.is.

3.1.5.3 Class.contextType

The contextType property on a class can be assigned a Context object created by React.createContext(). Using this property lets you consume the nearest current value of that Context type using this.context. You can reference this in any of the lifecycle methods including the render function.

If you are using the experimental public class fields syntax, you can use a static class field to initialize your contextType.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

3.1.5.4 Context.Consumer

A React component that subscribes to context changes. Using this component lets you subscribe to a context within a function component.

Requires a function as a child. The function receives the current context value and returns a React node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the defaultValue that was passed to createContext().

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

Creating Context and Provider

```
import React, {createContext} from 'react';
export const ThemeContext = createContext()
class ThemeContextProvider extends React.Component {
    constructor() {
        super()
        this.state = {
            isLightTheme: true,
            light: {color:'#555',ui:'#ddd',bg:'#eee'},
            dark: {color:'#ddd',ui:'#333',bg:'#555'}
        }
    }
    toggleTheme = () => {
        this.setState({isLightTheme:!this.state.isLightTheme})
    }
    render() {
        return(<ThemeContext.Provider value={{...this.state,
toggleTheme:this.toggleTheme}}>
            {this.props.children}
        </ThemeContext.Provider>)
    }
}
export default ThemeContextProvider
```

Consuming Context using Consumer

```
import React, { Component } from 'react';
import { ThemeContext } from './context/ThemeContext';
class Navbar extends Component {
    render() {
        return (
            <ThemeContext.Consumer>{(context)=>{
                const {isLightTheme, light, dark} = context
                const theme = isLightTheme ? light:dark;
                return(
                    <nav style={{background: theme.ui, color:theme.color}}>
                    <h1>Context App</h1>
                    <ul>
                        <li>Home</li>
                        <li>About</li>
                        <li>Contact</li>
                    </ul>
                </nav>
                )
            }}</ThemeContext.Consumer>
        );
    }
}
export default Navbar;
```

Consuming Context using contextType

```
import React, { Component } from 'react';
import { ThemeContext } from './context/ThemeContext';
class CourseList extends Component {
  static contextType = ThemeContext
  render() {
    const {isLightTheme, light, dark} = this.context
    const theme = isLightTheme ? light:dark;
    return (
      <div className="course-list" style={{color:theme.color,background:theme.bg}}>
        <ul>
          <li style={{background:theme.ui}}>React JS</li>
          <li style={{background:theme.ui}}>Angular</li>
          <li style={{background:theme.ui}}>Node</li>
        </ul>
      </div>
    );
  }
}

export default CourseList;
```

3.2 Functional Components

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

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

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

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

3.2.1 Props in Functional Component

We can pass props to a function-based component also just like we did in the class component. But to access a prop from a function we do not need to use the ‘this’ keyword anymore.

Functional components accept props as parameters and can be accessed directly. Below is the same example as above but this time using a function-based component instead of a class-based component.

We will pass some information as props from our Parent component to the Child component.

```
// Parent Component
function Parent{
        return(
                <div>
                    <h2>You are inside Parent Component</h2>
                    <Child name="User" userId = "5555"/>
                </div>
            );
    }
```

```
// Child Component
function Child (props){
        return(
                <div>
                    <h2>Hello, {props.name}</h2>
                    <h3>You are inside Child Component</h3>
                    <h3>Your user id is: {props.userId}</h3>
                </div>
            );
    }
```

```
import React from 'react';
import ReactDOM from 'react-dom';

//Index.js
ReactDOM.render(
    // passing props
    <Parent />,
    document.getElementById("root"));
```

3.2.2 Hooks

React hooks are JavaScript functions, which allows the developer to interact with the state and lifecycle methods through functional components

Earlier we could associate state, modify the state, and implement lifecycle methods only in class components, whereas functional components were used only to present the UI content, but now with React 16.8 we can achieve the above-mentioned tasks in functional components using React hooks.

Rules of Hooks:

- Only Call Hooks at the Top Level

Don’t call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That’s what allows React to correctly preserve the state of Hooks between multiple useState and useEffect calls.

- Only Call Hooks from React Functions

Don’t call Hooks from regular JavaScript functions. Instead, you can:

- ✓ Call Hooks from React function components.
- ✓ Call Hooks from custom Hooks

3.2.2.1 useState()

Making a change in DOM is typically done through the *useState()* in a functional component**.**

```
import React, { useState } from 'react';

function Example() {

  // Declare a new state variable, which we'll call "count"

  const [count, setCount] = useState(0);
```

What does calling useState do?

It declares a “state variable”. Our variable is called count. This is a way to “preserve” some values between the function calls — useState is a new way to use the exact same capabilities that this.state provides in a class. Normally, variables “disappear” when the function exits but state variables are preserved by React.

What do we pass to useState as an argument?

The only argument to the useState() Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. In our example, we just want a number for how many times the user clicked, so pass 0 as initial state for our variable. (If we wanted to store two different values in state, we would call useState() twice.).

What does useState return?

It returns a pair of values: the current state and a function that updates it. This is why we write const [count, setCount] = useState(). This is similar to this.state.count and this.setState in a class, except you get them in a pair.

We declare a state variable called count, and set it to 0. React will remember its current value between re-renders, and provide the most recent one to our function. If we want to update the current count, we can call setCount.

Reading State

```
<p>You clicked {count} times</p>
```

Updating State

```
<button onClick={() => setCount(count + 1)}>
    Click me
  </button>
```

Updating the state with a callback

When the new state is calculated using the previous state, you can update the state with a callback:

```
const [state, setState] = useState(initialState);

// changes state to `newState` and triggers re-rendering
setState(prevState => nextState);

// after re-render `state` becomes `newState`
Here are some use cases:

// Toggle a boolean
const [toggled, setToggled] = useState(false);
setToggled(toggled => !toggled);

// Increase a counter
const [count, setCount] = useState(0);
setCount(count => count + 1);

// Add an item to array
const [items, setItems] = useState([]);
setItems(items => [...items, 'New Item']);
```

Stale State

A closure is a function that captures variables from the outer scopes.

Closures (e.g. event handlers, callbacks) might capture state variables from the functional component scope. Because state variables change between renderings, closures should capture variables with the latest state value.

Otherwise, if the closure captures outdated state values, you might encounter stale state problem.

Let’s see how a stale state manifests itself. A component <DelayedCount> should count the number of button clicks, but with a delay of 3 seconds.

Here’s the first naive implementation:

```
function DelayedCount() {
  const [count, setCount] = useState(0);

  const handleClickAsync = () => {
    setTimeout(function delay() {
      setCount(count + 1);
    }, 3000);
  }

  return (
    <div>
      {count}
      <button onClick={handleClickAsync}>Increase async</button>
    </div>
  );
}
```

Open the demo. Click Increase async quickly a few times. count doesn’t reflect the actual number of clicks, some clicks are “eaten”.

React app multiple state

delay() is a stale closure that captures an outdated count variable from the initial render (when it was initialized with 0).

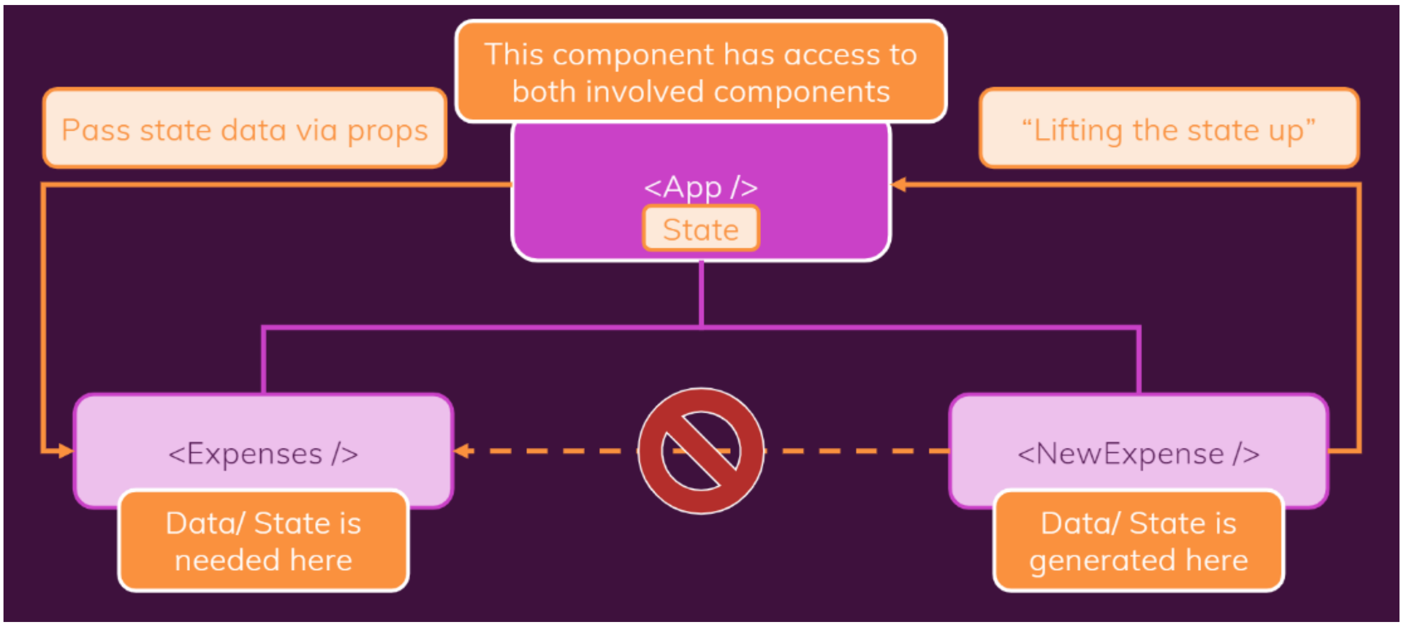
To fix the problem, let’s use a functional way to update count state:

```
function DelayedCount() {
  const [count, setCount] = useState(0);

  const handleClickAsync = () => {
    setTimeout(function delay() {
      setCount(count => count + 1);
    }, 3000);
  }

  return (
    <div>
      {count}
      <button onClick={handleClickAsync}>Increase async</button>
    </div>
  );}
```

Now setCount(count => count + 1) updates the count state correctly inside delay(). React makes sure the latest state value is supplied as an argument to the update state function. The stale closure is solved.

Lifting the state up

To share a state between two components, the most common operation is to move it up to their closest common ancestor(using props). This is called “lifting state up”.


This component has access to both involved components
Pass state data via props
“Lifting the state up”
<App />
State
<Expenses />
<NewExpense />
Data/ State is needed here
Data/ State is generated here

3.2.2.2 useRef()

The useRef is a hook that allows to directly create a reference to the DOM element in the functional component.

useRef is a function which accepts the initial value and returns a ref object whose .current property will be initialized with the initial value passed

useRef hook syntax

```
        let countRef = useRef(0)
        console.log(countRef.current)
```

The current property will be initialized with 0. The above code logs 0 to the console.

The countRef ref object returned is mutable and will persist for the full lifetime of the component

The most common use case for using refs is when we need to gain focus to the form elements.

```
import React, { useRef } from "react";
const CustomTextInput = () =>{
  const textInput = useRef(null);

  const focusTextInput = () => {
    textInput.current.focus();
  }
  return (
    <>
      Username:<input type="text" ref={textInput} /><br/><br/>
      <button onClick={focusTextInput}>Focus the text input</button>
    </>
  );
}
export default CustomTextInput
```

Line 4: textInput ref is created using useRef and value null is assigned
Line 12: assign the ref to the username input field
Line 13: Onclick of the button focusTextInput method is called
Line 7: The focus method is called to attain focus to the input field

```
import React, { useRef } from "react";
const Counter = () => {
  const counter = useRef(0);
  counter.current = counter.current + 1;

  return (
    <div>
      <h1>{counter.current}</h1>
    </div>
  );
};
export default Counter;
```

The above code renders always 1, because **updating the current property does not cause the component to re-render**.

We can access HTML element using refs. Suppose we have, a component Input which contains an input field or button element and the other components using the Input component might need access their DOM nodes (input and button element) for managing focus.

In such cases the refs can be forwarded to the child components

```
import React, { useRef, useEffect} from 'react';
import Input from './components/Input';
function App() {
  useEffect(()=>{
    firstNameRef.current.focus()
  },[])
  const firstNameRef = useRef(null)
  const lastNameRef = useRef(null)
  function firstNameKeyDown(e) {
    if(e.key === "Enter") {
      lastNameRef.current.focus()
    }
  }
  return (
    <div>
    <header>
     First Name: <Input ref={firstNameRef} placeholder="Enter first name here"
onKeyDown={firstNameKeyDown}/><br/><br/>
     Last Name: <Input ref={lastNameRef} placeholder="Enter last name here" /><br/><br/>
    </header>
  </div>
  );
}export default App;
```

Line 9 and 10 : The App component creates firstNameRef and lastNameRef
Line 19 and 20: The refs created are passed to the child component
Line 11: when enter key is pressed on the input the lastname field is focused

```
import React from "react";
const Input = React.forwardRef((props,ref) => (

     <input onKeyDown={props.onKeyDown} ref={ref} type="text"
placeholder={props.placeholder} style={props.style} />
))

export default Input
```

Line 3: The Input component uses React.forwardRef to fetch the ref passed. The ref is passed as a second argument to the **forwardRef** method
Line 5: The Input component passes the ref received to the input field

3.2.2.3 useImperativeHandle()

useImperativeHandle customizes the instance value that is exposed to parent components when using ref. As always, imperative code using refs should be avoided in most cases. useImperativeHandle should be used with forwardRef:

Parent Component

```
const App = () => {
  const ref = useRef();
  return (
    <div>
      <FancyInput ref={ref} />
      <button onClick={() => ref.current.focus()}>focus</button>
    </div>
  );
};
```

Child Component

```
import React, { useRef, forwardRef, useImperativeHandle } from "react";
import ReactDOM from "react-dom";

const FancyInput = forwardRef((props, ref) => {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} value="hello" />;
});
```

3.2.2.4 useEffect()

Why useEffect()?

In a class component, we have various life-cycle methods, to help us run some code after the component mounts or after the component updates. For example, componentDidMount and componentDidUpdate life-cycle methods.

As we cannot use these life-cycle methods in functional components, we can use the useEffect() hook instead.

What is useEffect()?

The useEffect() method is a hook provided by React so that we can use various life-cycle methods in functional components, without making it a class component.

Essentially speaking, The useEffect() hook lets us perform any side effects in functional components.

Examples of side-effects are fetch requests, manipulating DOM directly, using timer functions like setTimeout(), and more.

The component rendering and side-effect logic are independent. So it would be a mistake to perform side-effects directly in the body of the component.

useEffect() method gets executed in both cases i.e. after first render and every time the component gets updated.

We know that, we cannot use life-cycle methods in functional components, hence we can use useEffect() hook instead.

Do not forget that ***useEffect()*** method gets executed in both cases i.e. after first render and every time the component gets updated.

Syntax 1: useEffect() can be written as follows:

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

Syntax 2: The useEffect() method takes an array as a second argument

```
useEffect(() => {

//implementation

}, [])
```

If we pass an empty array as a second argument then the useEffect method runs only after the component mounts not after the component updates. It is *equivalent to* ***componentDidMount*** method in class based component.

When we pass a specific state as second argument as shown below, then the useEffect method will run once for the initial render and then every time when the specific state passed updates and not for other state updates.

```
useEffect(() => {
    //implementation
}, [age])
```

Async calls with useEffect() hook

We can handle asynchronous calls within the useEffect() method which is similar to how we handle asynchronous calls using componentDidMount() method in class based components.

We need to pass the second argument as an empty array to the useEffect() method, since we want the useEffect() method to be called only after the component mounts and not when the component updates.

Syntax:

```
useEffect(()=>{
// any asynchronous calls
},[])
```

Dependencies

- You DON'T need to add state updating functions : React guarantees that those functions never change, hence you don't need to add them as dependencies (you could though)
- You also DON'T need to add "built-in" APIs or functions like fetch(), localStorage etc (functions and features built-into the browser and hence available globally): These browser APIs / global functions are not related to the React component render cycle and they also never change
- You also DON'T need to add variables or functions you might've defined OUTSIDE of your components (e.g. if you create a new helper function in a separate file): Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function if those "things" could change because your component (or some parent component) re-rendered. That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Cleanup Function

Some side-effects need cleanup: close a socket, clear timers.

If the callback of useEffect(callback) returns a function, then useEffect() considers this as an *effect cleanup*:

```
useEffect(() => {
  // Side-effect...
  return function cleanup() {    // Side-effect cleanup...  };}, dependencies);
```

Cleanup works the following way:

A) After initial rendering, useEffect() invokes the callback having the side-effect. cleanup function is *not invoked*.

B) On later renderings, before invoking the next side-effect
callback, useEffect() *invokes* the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect.

C) Finally, after unmounting the component, useEffect() *invokes* the cleanup function from the latest side-effect.

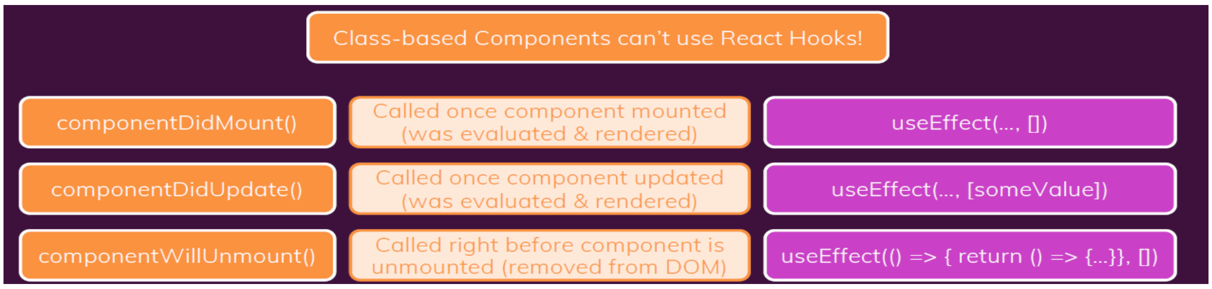
```
import React, { useEffect, useState } from 'react';
export default function App() {
  const [state, setState] = useState(null);
  useEffect(() => {
    console.log('I am the effect');
    return () => {
      console.log('I run after re-render, but before the next useEffect');
    };
  });
  console.log('I am just part of render');
  return (
    <>
      <button
        onClick={() => {
          setState('Some v. important state.');
        }}
      >
        Click me
      </button>
      <p>state: {state}</p>
    </>
  );
}
```

When the above component first renders, in the console you see:

```
> I am just part of render

> I am the effect
```

If you then click the button (triggering a re-render), the following lines are printed underneath:

```
> I am just part of render

> I run after re-render, but before the next useEffect

> I am the effect
```

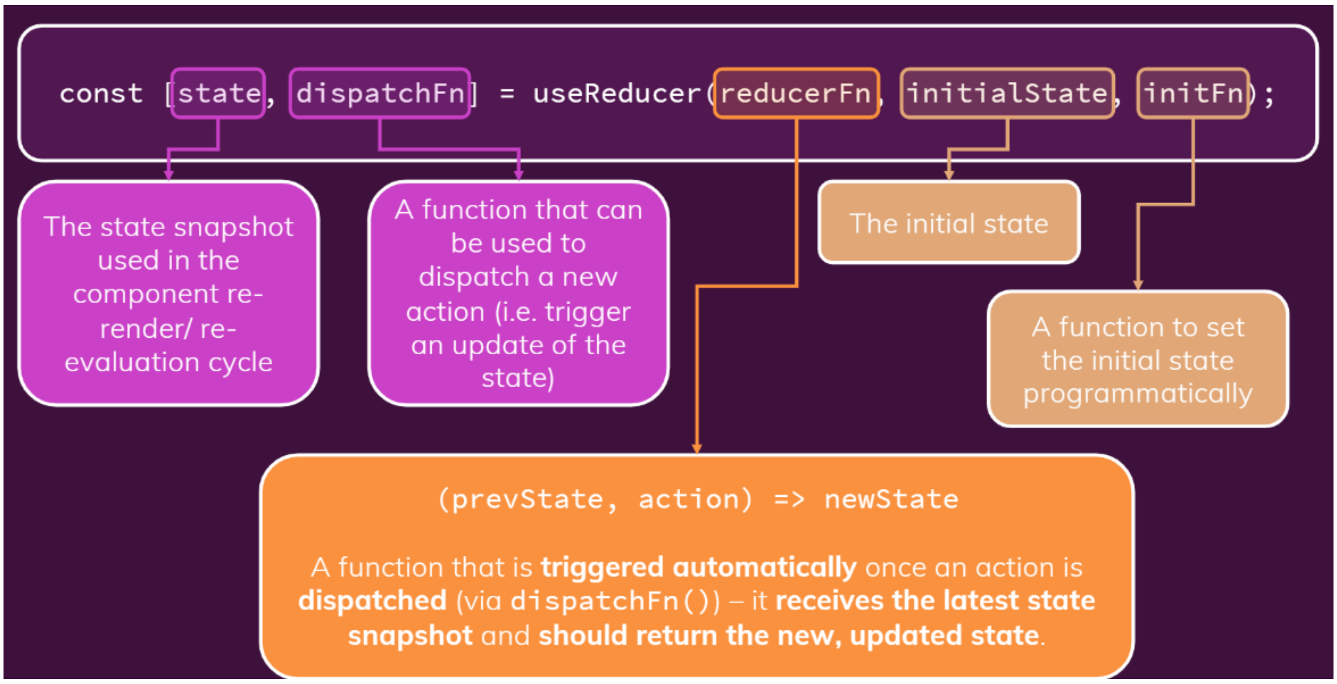


Class-based Components can't use React Hooks!

componentDidMount()	Called once component mounted (was evaluated & rendered)	useEffect(..., [])
componentDidUpdate()	Called once component updated (was evaluated & rendered)	useEffect(..., [someValue])
componentWillUnmount()	Called right before component is unmounted (removed from DOM)	useEffect(() => { return () => {...}}, [])

3.2.2.5 useReducer()

The useReducer hook is an alternative for useState. It can be used when we have complex state logic that involves multiple sub-values or when the next state depends on the previous state.


const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);
The state snapshot used in the component re-render/ re-evaluation cycle
A function that can be used to dispatch a new action (i.e. trigger an update of the state)
The initial state
A function to set the initial state programmatically
(prevState, action) => newState
A function that is **triggered automatically** once an action is **dispatched** (via dispatchFn()) – it **receives the latest state snapshot** and **should return the new, updated state**.

Note:

React guarantees that dispatch function identity is stable and won't change on re-renders. This is why it's safe to omit from the useEffect or useCallback dependency list.

Points to note about useReducer

- useReducer is a hook which is pre-built with React and used with React and not with Redux
- useReducer behaves like redux only for local state management and not for global state management
- Using useReducer hook with useContext hook, we can also do global state management

The useReducer hook accepts two arguments

- The first argument is a reducer function that takes current state and action object as arguments and returns the new modified state
- The second argument is the initial state

The useReducer hook will return the current state and dispatch method similar to useState

The dispatch method can be used to dispatch an action

Here’s the counter example using a reducer:

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

3.2.2.6 useContext()

Accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree.

```
const value = useContext(MyContext);
```

When the nearest <MyContext.Provider> above the component updates, this Hook will trigger a rerender with the latest context value passed to that MyContext provider. Even if an ancestor uses React.memo or shouldComponentUpdate, a rerender will still happen starting at the component itself using useContext.

Accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree.

Note:

useContext(MyContext) is equivalent to static contextType = MyContext in a class, or to <MyContext.Consumer>.

useContext(MyContext) only lets you read the context and subscribe to its changes. You still need a <MyContext.Provider> above in the tree to provide the value for this context.

Simple example creating a context

```
import React, { createContext } from ‘react’;
import ReactDOM from ‘react-dom’;
const MessageContext = createContext();
myApp=()=>{
  return (
    <MessageContext.Provider value=”hello”>
      <div>
        <Test/>
      </div>
    </MessageContext.Provider>
  );
}
```

In child component Test we can access the message value as below –

```
Import { useContext } from ‘react’;
Test =()=>{
  const messageValue=useContext(MessageContext);
  return (
    <div>Message: {messageValue} </div>
  );
}
```

3.2.2.7 useCallBack()

React.memo is a higher order component.

If your component renders the same result given the same props, you can wrap it in a call to React.memo for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

```
React.memo
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

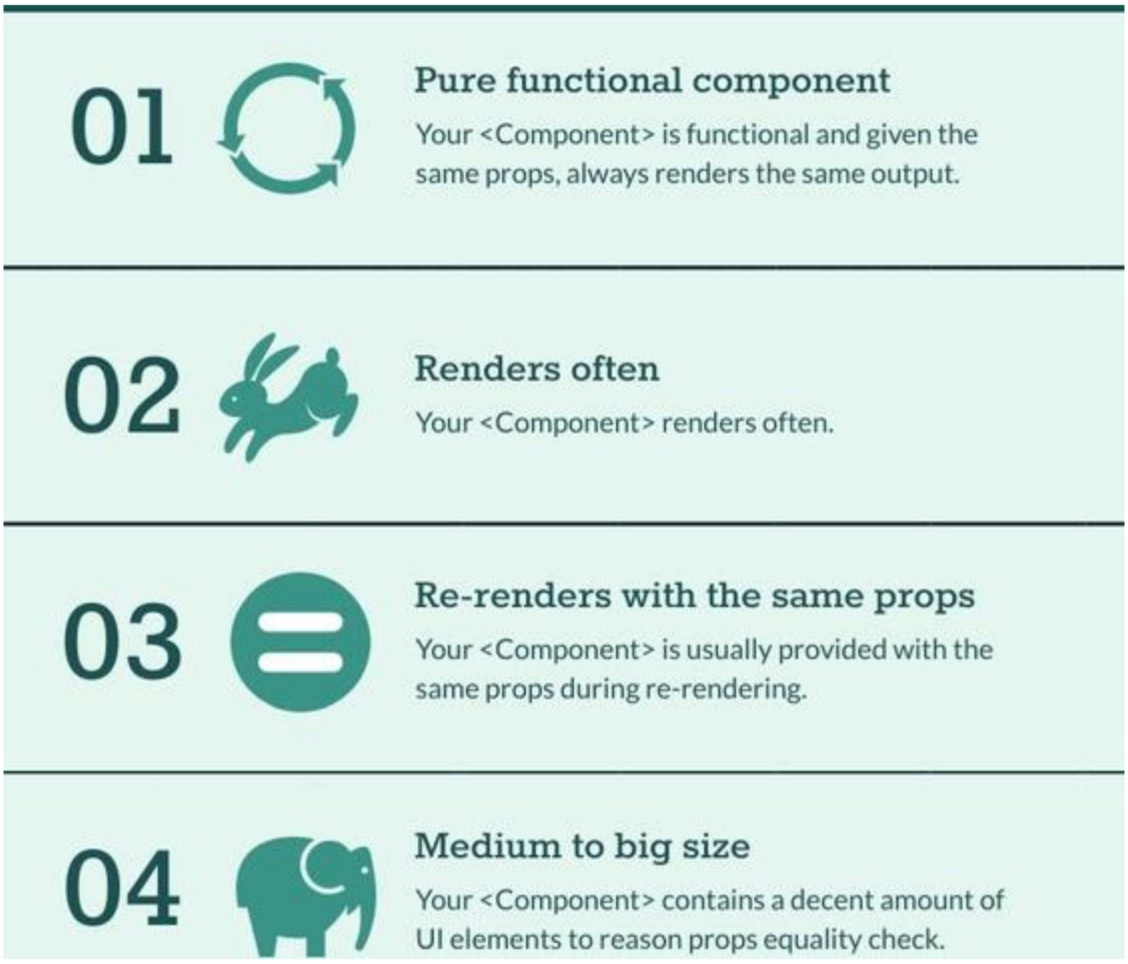
React.memo only checks for prop changes. If your function component wrapped in React.memo has a useState, useReducer or useContext Hook in its implementation, it will still rerender when state or context change.

By default it will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.

```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}
```

export default React.memo(MyComponent, areEqual);
This method only exists as a performance optimization. Do not rely on it to “prevent” a render, as this can lead to bugs.

When to use React.memo?



01 **Pure functional component**

Your <Component> is functional and given the same props, always renders the same output.

02 **Renders often**

Your <Component> renders often.

03 **Re-renders with the same props**

Your <Component> is usually provided with the same props during re-rendering.

04 **Medium to big size**

Your <Component> contains a decent amount of UI elements to reason props equality check.

What is the cost of React.memo?
A memoized component compares old with news props to decide, if to re-render - each render cycle.
A plain component does not care and just renders, after props/state change in a parent.

When NOT use React memo?

There are no hard rules. Things, that affect React.memo negatively:

1. component often re-renders with props, that have changed anyway
2. component is cheap to re-render
3. comparison function is expensive to perform

```
useCallback(fn, [deps])
```

Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. shouldComponentUpdate).

```
import React, { useState } from "react";
import MemoCounter from './MemoCounter';
const Memo
Parent = () => {

   const [count1, setCount1] = useState(0)
   const [count2, setCount2] = useState(0)

   const increaseCounter1 = React.useCallback(() => {
      setCount1(count1 => count1 + 1)
   },[])
   const increaseCounter2 = React.useCallback(() => {
      setCount2(count2 => count2 + 1)
   },[])
   return (<div>
      <MemoCounter onClick={increaseCounter1} value={count1}>Counter 1</MemoCounter>
      <MemoCounter onClick={increaseCounter2} value={count2}>Counter 2</MemoCounter>
   </div>)
}
export default MemoParent;
```

Since empty array is passed as a second argument to the useCallback, the increaseCounter1 and increaseCounter2 functions gets created only once.

So the second Counter Component doesn’t gets re-rendered.

```
import React from "react";
const MemoCounter = ({value,children,onClick}) => {
   console.log('Render',children)
   return(<button onClick={onClick}>
      {children}: {value}
   </button>)
}
export default React.memo(MemoCounter);
```

Imagine you have a component that renders a big list of items:

```
import useSearch from './fetch-items';
function MyBigList({ term, onItemClick }) {
  const items = useSearch(term);
  const map = item => <div onClick={onItemClick}>{item}</div>;
 return <div>{items.map(map)}</div>;
}

export default React.memo(MyBigList);
```

The list could be big, maybe hundreds of items. To prevent useless list re-renderings, you wrap it into React.memo().

The parent component of MyBigList provides a handler function to know when an item is clicked:

```
import { useCallback } from 'react';

export default function MyParent({ term }) {
  const onItemClick = useCallback(event => {
    console.log('You clicked ', event.currentTarget);
  }, [term]);

  return (
    <MyBigList
      term={term}
      onItemClick={onItemClick}
    />
  );
}
```

onItemClick callback is memoized by useCallback(). As long as term is the same, useCallback() returns the same function object.

When MyParent component re-renders, onItemClick function object remains the same and doesn’t break the memoization of MyBigList.

3.2.2.8 useMemo()

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Returns a memoized value.

Pass a “create” function and an array of dependencies. useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

Remember that the function passed to useMemo runs during rendering. Don’t do anything there that you wouldn’t normally do while rendering. For example, side effects belong in useEffect, not useMemo.

If no array is provided, a new value will be computed on every render.

In other words, useCallback gives you referential equality between renders for functions. And useMemo gives you referential equality between renders for values.

```
import React, {useState, useMemo} from 'react';
```

```
function App() {
  const [length, set_length] = useState(3);
  const [name, set_name] = useState('John Doe');

  return (
    <>
      <input value={name} onChange={e => set_name(e.target.value)} />
      <NameDisplay name={name}/>
      <hr />
      <input value={length} onChange={e => set_length(Number(e.target.value))} />
      <FibDisplay length={length} />
    </>
  );
}

function FibDisplay({length}) {
  const numbers = useMemo(() => {
    console.log('Calculating numbers...');
    const result = [1, 1];
    for (let i = 2; i < length; i++) {
      result[i] = result[i - 1] + result[i - 2];
    }
    return result;
  }, [length]);

  return <p>{length} numbers of the fibonacci sequence: {numbers.join(', ')}</p>;
}

const NameDisplay = React.memo(function ({name}) {
  console.log('Rerendering name...');
  return <p>Your name is {name}</p>;
});

export default App;
```

3.2.2.9 Custom Hooks

When we want to share logic between two JavaScript functions, we extract it to a third function. Both components and Hooks are functions, so this works for them too!

Custom Hooks helps developers to create functionality that can be reused across multiple components.

Why not create regular functions and reuse them. Why do we need custom hooks

We need custom hooks because a custom hook uses other hooks and using which we will be able to associate the state and component life cycle methods to a functional component. If we do not use any other hook within a custom hook then it is a normal function.

A custom Hook is a JavaScript function whose name starts with “use” and that may call other Hooks.

The need for Custom Hooks:

The main reason for which you should be using Custom hooks is to maintain the concept of DRY(Don't Repeat Yourself) in your React apps. For example, suppose you have some logic that makes use of some built-in hooks and you need to use the logic in multiple functional components. So, there are two ways left for you — 1) write the same logic in each and every component (which violates the concept of DRY) and 2) create a separate function that wraps the logic inside it and then call it from those components. The second option is undoubtedly a better choice as you have to write the logic only once. Here, the separate function you created is the custom hook. So, whenever you feel that you have a logic that is to be used in multiple functional components(hooks don't work in class components), just create a separate custom hook for it and use it.

Building a custom hook:

Creating a custom hook is the same as creating a JavaScript function whose name starts with "use". It can use other hooks inside it, return anything you want it to return, take anything as parameters.

// useFetch.js

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

export default function useFetch(url) {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return data;
}
```

This is how you would use the custom hook:

```
import React from "react";
import useFetch from "./useFetch";

export default function DataLoader(props) {
  const data = useFetch("http://localhost:3001/links/");
  return (
    <div>
      <ul>
        {data.map(el => (
          <li key={el.id}>{el.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

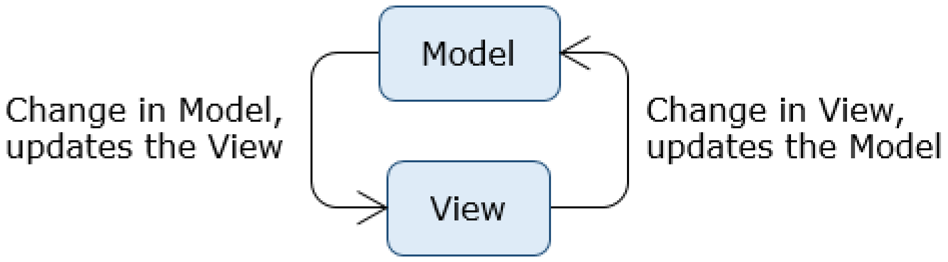
3.3 Forms in Class & Functional Components

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

- The form that we create using React to accept user inputs is different from HTML forms
- Users will not be able to input value in the form input field because ReactJS uses unidirectional data flow

Before discussing Forms in React, let us first understand the unidirectional data flow.

Most of the frameworks follow two-way data binding. This would allow for updates from both the end i.e. changes can be done from view to model and vice versa.


Model
Change in Model, updates the View
Change in View, updates the Model
View

In this approach state of the model could be mutated by both model and view, it would cause unpredictable data flow.

React uses unidirectional data flow pattern wherein the changes will be done only from component to view but not vice versa.

This would prevent unpredictable data flow and easy to debug.

3.3.1 Controlled Components

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with **setState()** in class components and state updating function obtained from **useState()** hook in functional components.

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

Below is an example of a controlled component where it has a value attribute and a handler function:

```
<input type="text" value={this.state.name} onChange={this.handleData}/>
```

3.3.2 Uncontrolled Components

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Below is the example for an uncontrolled component:

```
        <input type="text" />
```

Here, input will not have value attribute and event handler, instead to get the value from the DOM ref is used as follows.

```
<input type="text" ref={this.input}/>
```

In React, ref provides a way to access the DOM node or the React elements, and helps in getting the value from the DOM.

refs are to be used when we have the below requirements:

- Integrating with third-party libraries, like D3 which has it's own DOM manipulation API's
- To use media playbacks like audio, video. refs provide control over the DOM API.
- Trigger imperative animations

By using refs, we are deviating from the React way of handling data. In React we store data as state and when the state in updated component gets re-rendered. This follows a unidirectional data flow and the data is stored in a single source of truth i.e., state.

Let's see how ref and refs will be used to get the reference of a node in class components and for functional components useRef() hook as mentioned earlier.

We need to create a ref instance within the constructor as shown below:

```
constructor(props) {

   super(props);

   this.nameRef = React.createRef();

}
```

We need to assign the ref instance created in the constructor to the input field using ref attribute, as shown below:

```
        <input type="text" ref={this.nameRef} />
```

When we assign a ref attribute to the input field, the ref created in the constructor will receive the DOM element.

We can access the value of the input field as shown below:

```
            this.nameRef.current.value;
```

3.3.3 Controlled Vs Uncontrolled Component

CONTROLLED COMPONENT	**UNCONTROLLED COMPONENT**
Does not maintain its internal state.	Maintains its internal state.
Data is controlled by the parent component.	Data is controlled by the DOM itself.
Accepts its current value as a prop.	Uses a ref for their current values.
Allows validation control.	Does not allow validation control.
Better control over the form elements and data.	Limited control over the form elements and data.

4. Styling

4.1 Styling components using CSS inline styling

In React, components will have inline styles and it should be specified as an object as shown below

<h1 style={{color:'green'}}>Welcome to React</h1>

The CSS properties has to be mentioned within an object and the object has to be provided as a value to the style property within an expression. The 2 pairs of curly braces are used, where the first pair is the object with style properties and second pair is the JSX expression specified to the style property.

When we have CSS properties with multiple words like background-color, font--family etc, in React, those properties should be camelCased by removing hypen as shown below.

```
<h1 style={{color:'green',fontFamily:'verdana'}}>Welcome to React</h1>
```

4.2 Styling components using CSS modules

A CSS Module is a CSS file in which all class names and animation names are scoped locally by default.

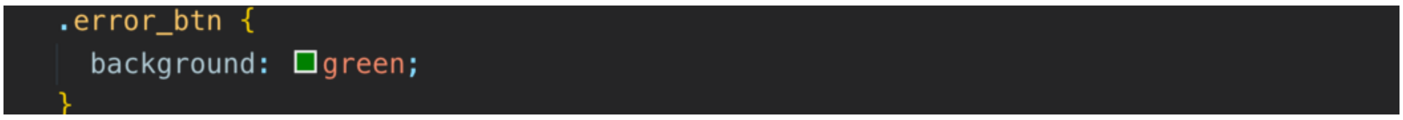
CSS Modules let you write styles in CSS files but consume them as JavaScript objects for additional processing and safety. CSS Modules are very popular because they automatically make class and animation names unique so you don't have to worry about selector name collisions.

That's the very simple premise of CSS Modules:

Each React component gets its own CSS file, which is scoped to that file and component.

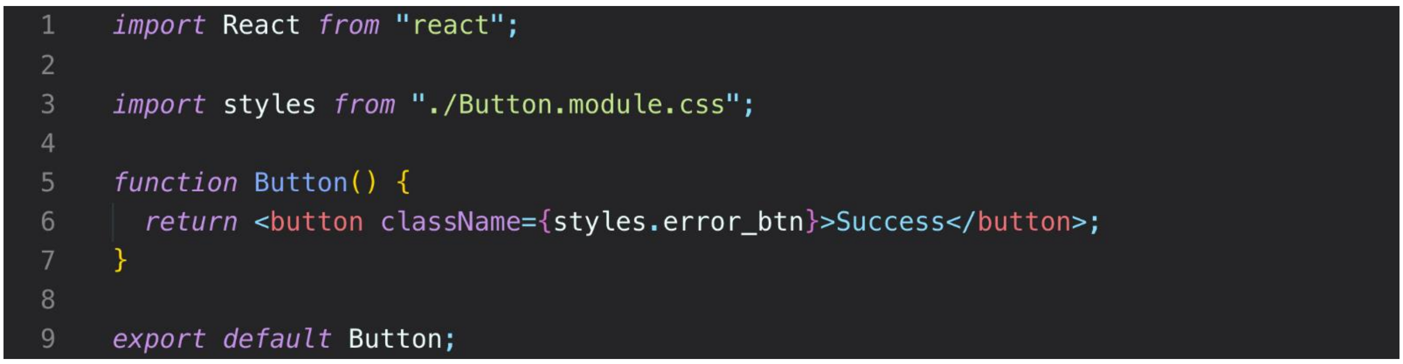
This project supports CSS Modules alongside regular stylesheets using the **[name].module.css** file naming convention

Tip: Should you want to preprocess a stylesheet with Sass then make sure to follow the installation instructions and then change the stylesheet file extension as follows: **[name].module.scss or [name].module.sass.**



```css
.error_btn {
  background: green;
}
```

Button.module.css



```jsx
import React from "react";

import styles from "./Button.module.css";

function Button() {
  return <button className={styles.error_btn}>Success</button>;
}

export default Button;
```

Button.js

For media queries

```
@media (max-width: 768px) {
        .error_btn {
        color: red;
        }
        }
```

4.3 Styling components using styled components

styled-components is the result of wondering how we could enhance CSS for styling React component systems. By focusing on a single use case we managed to optimize the experience for developers as well as the output for end users.

Apart from the improved experience for developers, styled-components provides:

- **Automatic critical CSS**: styled-components keeps track of which components are rendered on a page and injects their styles and nothing else, fully automatically. Combined with code splitting, this means your users load the least amount of code necessary.
- **No class name bugs**: styled-components generates unique class names for your styles. You never have to worry about duplication, overlap or misspellings.
- **Easier deletion of CSS**: it can be hard to know whether a class name is used somewhere in your codebase. styled-components makes it obvious, as every bit of styling is tied to a specific component. If the component is unused (which tooling can detect) and gets deleted, all its styles get deleted with it.
- **Simple dynamic styling**: adapting the styling of a component based on its props or a global theme is simple and intuitive without having to manually manage dozens of classes.
- **Painless maintenance**: you never have to hunt across different files to find the styling affecting your component, so maintenance is a piece of cake no matter how big your codebase is.
- **Automatic vendor prefixing**: write your CSS to the current standard and let styled-components handle the rest.

You get all of these benefits while still writing the CSS you know and love, just bound to individual components.

Installation

```
npm install --save styled-components
```

First, let's import styled-components

import styled from 'styled-components'

styled-components utilises tagged template literals to style your components.

It removes the mapping between components and styles. This means that when you're defining your styles, you're actually creating a normal React component, that has your styles attached to it.

This example creates two simple components, a wrapper and a title, with some styles attached to it:

// Create a Title component that'll render an <h1> tag with some styles

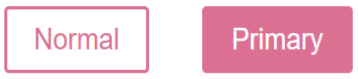
```
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

// Create a Wrapper component that'll render a <section> tag with some styles
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;

// Use Title and Wrapper like any other React component – except they're styled!
render(
  <Wrapper>
    <Title>
      Hello World!
    </Title>
  </Wrapper>
);
```

Adapting based on props

You can pass a function ("interpolations") to a styled component's template literal to adapt it based on its props.

This button component has a primary state that changes its color. When setting the primary prop to true, we are swapping out its background and text color.

```
const Button = styled.button`
  /* Adapt the colors based on primary prop */
  background: ${props => props.primary ? "palevioletred" : "white"};
  color: ${props => props.primary ? "white" : "palevioletred"};

  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;

render(
  <div>
    <Button>Normal</Button>
    <Button primary>Primary</Button>
  </div>
);
```



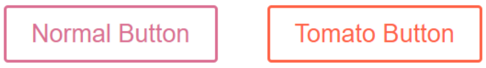
Extending Styles

Quite frequently you might want to use a component, but change it slightly for a single case. Now, you could pass in an interpolated function and change them based on some props, but that's quite a lot of effort for overriding the styles once.

To easily make a new component that inherits the styling of another, just wrap it in the styled() constructor. Here we use the button from the last section and create a special one, extending it with some color-related styling:

// The Button from the last section without the interpolations

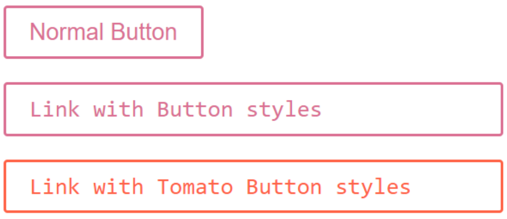
```
const Button = styled.button`
  color: palevioletred;
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;
// A new component based on Button, but with some override styles
const TomatoButton = styled(Button)`
  color: tomato;
  border-color: tomato;
`;
render(
  <div>
    <Button>Normal Button</Button>
    <TomatoButton>Tomato Button</TomatoButton>
  </div>);
```



In some cases you might want to change which tag or component a styled component renders. This is common when building a navigation bar for example, where there are a mix of anchor links and buttons but they should be styled identically.

For this situation, we have an escape hatch. You can use the "as" polymorphic prop to dynamically swap out the element that receives the styles you wrote:

```
const Button = styled.button`
  display: inline-block;
  color: palevioletred;
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
  display: block;
`;

const TomatoButton = styled(Button)`
  color: tomato;
  border-color: tomato;
`;

render(
  <div>
    <Button>Normal Button</Button>
    <Button as="a" href="#">Link with Button styles</Button>
    <TomatoButton as="a" href="#">Link with Tomato Button styles</TomatoButton>
  </div>
);
```



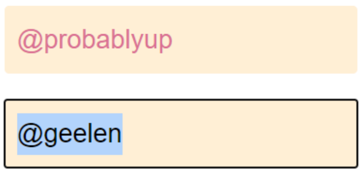
This works perfectly fine with custom components too!

Passed props

If the styled target is a simple element (e.g. styled.div), styled-components passes through any known HTML attribute to the DOM. If it is a custom React component (e.g. styled(MyComponent)), styled-components passes through all props.

This example shows how all props of the Input component are passed on to the DOM node that is mounted, as with React elements.

```
// Create an Input component that'll render an <input> tag with some styles
const Input = styled.input`
  padding: 0.5em;
  margin: 0.5em;
  color: ${props => props.inputColor || "palevioletred"};
  background: papayawhip;
  border: none;
  border-radius: 3px;
`;

// Render a styled text input with the standard input color, and one with a custom input color
render(
  <div>
    <Input defaultValue="@probablyup" type="text" />
    <Input defaultValue="@geelen" type="text" inputColor="rebeccapurple" />
  </div>
);
```



Note how the inputColor prop is not passed to the DOM, but type and defaultValue are. That is styled-components being smart enough to filter non-standard attributes automatically for you.

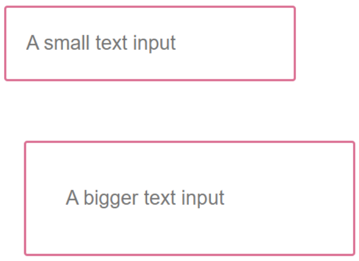
Attaching additional props

To avoid unnecessary wrappers that just pass on some props to the rendered component, or element, you can use the [.attrs constructor](). It allows you to attach additional props (or "attributes") to a component.

This way you can for example attach static props to an element, or pass a third-party prop like activeClassName to React Router's Link component. Furthermore you can also attach more dynamic props to a component. The .attrs object also takes functions, that receive the props that the component receives. The return value will be merged into the resulting props as well.

Here we render an Input component and attach some dynamic and static attributes to it:

```
const Input = styled.input.attrs(props => ({
  // we can define static props
  type: "text",

  // or we can define dynamic ones
  size: props.size || "1em",
}))`
  color: palevioletred;
  font-size: 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;

  /* here we use the dynamically computed prop */
  margin: ${props => props.size};
  padding: ${props => props.size};
`;

render(
  <div>
    <Input placeholder="A small text input" />
    <br />
    <Input placeholder="A bigger text input" size="2em" />
  </div>
);
```



Overriding .attrs

Notice that when wrapping styled components, .attrs are applied from the innermost styled component to the outermost styled component.

This allows each wrapper to **override** nested uses of .attrs, similarly to how css properties defined later in a stylesheet override previous declarations.

Input's .attrs are applied first, and then PasswordInput's .attrs:

```
const Input = styled.input.attrs(props => ({
  type: "text",
  size: props.size || "1em",
}))`
  border: 2px solid palevioletred;
  margin: ${props => props.size};
  padding: ${props => props.size};
`;

// Input's attrs will be applied first, and then this attrs obj
const PasswordInput = styled(Input).attrs({
  type: "password",
})`
  // similarly, border will override Input's border
  border: 2px solid aqua;
`;

render(
  <div>
    <Input placeholder="A bigger text input" size="2em" />
    <br />
    {/* Notice we can still use the size attr from Input */}
    <PasswordInput placeholder="A bigger password input" size="2em" />
  </div>
)
```

This is why PasswordInput is of a password type, but still uses the size attribute from Input.

The ampersand (&) can be used to refer back to the main component. Here are some more examples of its potential usage:

```
const Thing = styled.div.attrs((/* props */) => ({ tabIndex: 0 }))`
  color: blue;

  &:hover {
    color: red; // <Thing> when hovered
  }

  & ~ & {
    background: tomato; // <Thing> as a sibling of <Thing>, but maybe not directly next to it
  }

  & + & {
    background: lime; // <Thing> next to <Thing>
  }

  &.something {
    background: orange; // <Thing> tagged with an additional CSS class ".something"
  }

  .something-else & {
    border: 1px solid; // <Thing> inside another element labeled ".something-else"
  }
`


render(
  <React.Fragment>
    <Thing>Hello world!</Thing>
    <Thing>How ya doing?</Thing>
    <Thing className="something">The sun is shining...</Thing>
    <div>Pretty nice day today.</div>
    <Thing>Don't you think?</Thing>
    <div className="something-else">
      <Thing>Splendid.</Thing>
    </div>
  </React.Fragment>
)
```

4.4 Styling components using CSS file

The index.css file is imported in the index.js file

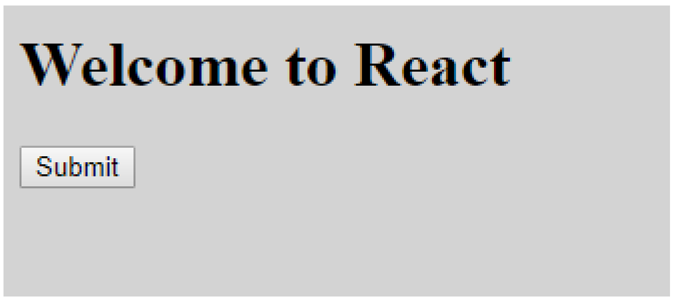
```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
```

So any styling written in the index.css file can be used in our components.

Remove the styling provided in the index.css file and add below code

```
body {
  background-color:lightgrey;
}
```

Now run the application, you can observe the background-color is grey when App component is rendered, as shown below



index.css is like a global CSS file for the application, any styling that is written within that file will be applied to all the components, since it is imported within the index.js file

If you want to create a specific CSS file for your component, create a css file and import the CSS file only within that component.

The CSS class can be applied to any JSX element using className attribute as shown below

```
<h1 className="highlight">Heading</h1>
```

Modify the CSS file App.css within the src folder and create a class button within App.css file as shown below

```
.button {
  background-color:blue;
  color:white;
  border-radius:10px;
  width:100px;
  height:30px;
}
```

Import the App.css file within the App component and apply the button class to the button element as shown below:

```
import './App.css';
<button className="button">Submit</button>
```

4.5 Styling components using Bootstrap

To style components using the bootstrap library, we need to install the bootstrap library

```
npm install bootstrap
```

Once you install bootstrap, the bootstrap CSS file will be present within the node_modules folder

import the bootstrap CSS file within the AppComp component and apply the bootstrap btn btn-success class to the button as shown below:

```
import 'bootstrap/dist/css/bootstrap.min.css';
<button className="btn btn-success">Submit</button>
```

4.6 Styling components using react-bootstrap

React-Bootstrap is a library of re-useable front-end components. We can have the look-and-feel of Twitter Bootstrap, but with much cleaner code, via Facebook's React JS framework.

React-bootstrap can be installed by running the below command

```
npm install react-bootstrap bootstrap.
```

Whenever any react-bootstrap component is used in an application, the corresponding react-bootstrap library should be imported

For example, if react-bootstrap button is created, then the corresponding bootstrap library should be imported as follows:

```
import Button from 'react-bootstrap/Button';
<Button variant="warning"> Click here </Button>
```

4.7 Styling components using Material UI

Material-UI is a npm package that helps in creating intuitive and beautiful React components.

It was developed by Google with two goals in mind :

1. To Synthesize principles of good design
2. To provide a unified experience across devices

The Material UI components work independent of any other stylesheets, i.e., they are self-supporting.

Need of Material UI

Material UI provides various pre-coded components such as Card, Grid, Dialog, CssBaseline etc that makes your web application more user friendly.

It also makes up a good visual language that is compatible with the users.

Material-UI is a popular React UI framework which provides various components and themes for styling React components.

Material-UI can be installed by running the below command

```
npm install @material-ui/core
```

Whenever any Material-UI component is used in an application, the corresponding Material-UI library should be imported. For example to style buttons Material-UI provides Button component, which has to be imported from @material-ui/core/Button library. The components provided by Material-UI would be present inside node_modules

```
import Button from '@material-ui/core/Button';
```

Installation of Material UI

The following command is used to install material-ui component

```
npm install material-ui/core --save
```

Installation of Material UI icons

The following command is used to install material-ui icons

```
npm install @material-ui/icons --save
```

Steps to import the Material UI Components

For example, consider a requirement of using a Material UI button.

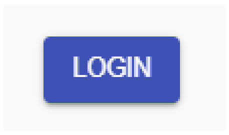
The following is the way to import the Button

```
import Button from ‘@material-ui/core/Button’
```

Steps to use the Components

The following is the syntax to render a button using Material UI.

```
<Button color="primary" variant="contained"> Login </Button>
```

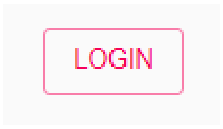


There can be different **attributes** for different Components. For Example, a button has attributes like variant, color, fullWidth, href, size etc that determines different properties of it.

Here color attribute can also take a different value. Say, color=”secondary” changes the color of the button as Pink

Similarly, the attribute variant can also take other values. Say, variant=”outlined” that gives an outline to the button based on the color attribute.

A button with color=”secondary” and variant=”outlined” appears as shown below



5. React Portals

In React, portals can be used to render an element outside of its parent component's DOM node while preserving its position in the React hierarchy, allowing it to maintain the properties and behaviors it inherited from the React tree.

```
ReactDOM.createPortal(child, container)
```

The first argument (child) is any renderable React child, such as an element, string, or fragment. The second argument (container) is a DOM element.

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually "break out" of its container. For example, dialogs, hovercards, and tooltips.

App.js

```
import React, {Component} from 'react';
import './App.css'
import PortalDemo from './PortalDemo.js';

class App extends Component {
   render () {
      return (
         <div className='App'>
       <PortalDemo />
   </div>
      );
   }
}
export default App;
```

PortalDemo.js

```
import React from 'react'
import ReactDOM from 'react-dom'

function PortalDemo(){
   return ReactDOM.createPortal(
     <h1>Portals Demo</h1>,
     document.getElementById('portal-root')
   )
}
export default PortalDemo
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>It is required to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <div id="portal-root"></div>
  </body>
</html>
```

Now, open the Inspect (ctrl + shift + I). In this window, select the Elements section and then click on the <div id="portal-root"></div> component. Here, we can see that each tag is under the "portal-root" DOM node, not the "root" DOM node.

Hence, we can see that how React Portal provides the ability to break out of root DOM tree.

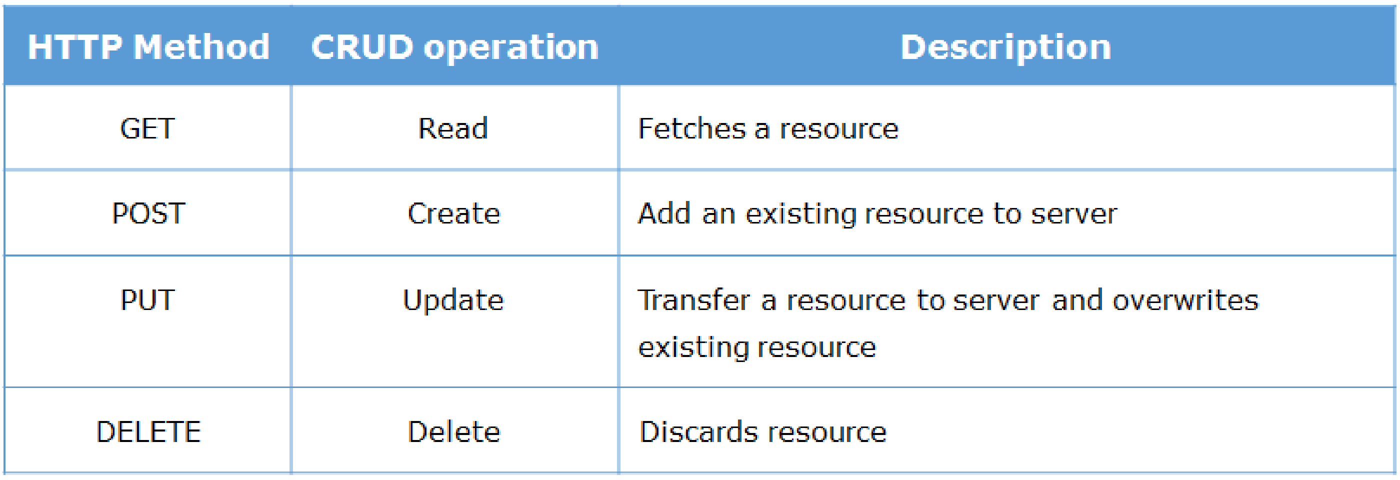
6. HTTP Requests

Applications communicate with one or more web services to consume data, update data, or display dynamic data. Data can change based on which user is logged in, or the location from where the user logged in, and correspondingly show different results on the webpage.

These web services have RESTful APIs that can easily be consumed in an application by using HTTP requests. React application use HTTP to communicate the user's interaction to the web services and to return the requested content for the user to browse.

When the user clicks on a link, a new HTTP request is generated to fetch the required content from the server. Once the requested content has been found, the server then generates a response. This response contains the content. The user will be able to see or interact with the content once it has been rendered.

As you have seen earlier, HTTP has 4 main methods that we are familiar with: GET, POST, PUT and DELETE.



HTTP Method	CRUD operation	Description
GET	Read	Fetches a resource
POST	Create	Add an existing resource to server
PUT	Update	Transfer a resource to server and overwrites existing resource
DELETE	Delete	Discards resource

There are several libraries that can help handle HTTP functionalities, for example:

- Fetch: This API has been a developer favorite, thanks to the familiarity of XMLHttpRequest combined with a more powerful and flexible feature set.
- Axios: While it is similar to the Fetch API, Axios prefers using promises and is based on the $http service within Angular.js.

Axios has some advantages over Fetch:

- It can automatically convert a response to JSON
- A request can be cancelled
- It has wide range of browser support.

6.1 Fetch

The Fetch API provides a fetch() method defined on the window object, which you can use to perform requests. This method returns a Promise that you can use to retrieve the response of the request.

The fetch method only has one mandatory argument, which is the URL of the resource you wish to fetch.

It returns a Promise that resolves to Response if the fetch operation is successful or throws an error that can be caught in case the fetch fails. You can also optionally pass in an init options object as the second argument.

The Promise returned from fetch() **won't reject on HTTP error status** even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with ok status set to false), and it will only reject on network failure or if anything prevented the request from completing.

Syntax:

PromiseReturned = fetch(urlOfTheSite, [options])

- urlOfTheSite – The URL to be accessed.
- options – optional parameters: method, headers, etc.

Without options, this is a simple/default GET request which downloads the contents from the URL. The fetch() returns a promise which needs to be resolved to obtain the response from the server or for handling the error.

Getting a response from a fetch() is a two-step process.

1. The promise object returned by fetch() needs to be resolved to an object after the server sends a response.

Here, HTTP status needs to be checked to see it is successful or not.
The promise will be rejected if the fetch is unable to make a successful HTTP-request to the server e.g. may be due to network issues, or if the URL mentioned in fetch does not exist.
HTTP-status can be seen in response properties easily by doing console.log

- status – HTTP status code returned from a response, e.g. 200.
- ok – Boolean, true if the HTTP status code returned from a response, is 200-299.

2. Get the response body using additional methods.

Response object can be converted into various formats by using multiple methods to access the body/data from response object:

- response.text() –read body/data from response object as a text,
- response.json() – parse body/data from response object as JSON,
- response.formData() – return body/data from response object as FormData
- response.blob() – return body/data from response object as Blob (binary data with it's type)

//pass any url that you wish to access to fetch()

```
let response = await fetch(url);
if (response.ok) { // if HTTP-status is 200-299
   // get the response body
   let json = await response.json();
   console.log(json)
}
else {
   console.log("HTTP-Error: " + response.status);
}
```

To send a POST request, use the following code:

```
const params = {
   param1: value1,
   param2: value2;
};
const options = {
   method: 'POST',
   headers: {
      "Content-type": "application/json
   },

   body: JSON.stringify( params )
};
fetch( 'https://domain.com/path/', options )
   .then( response => response.json() )
   .then( response => {
      // Do something with response.
   } );
```

6.2 Axios

Axios is a promise-based HTTP Client for node.js and the browser. It is isomorphic (= it can run in the browser and nodejs with the same codebase). On the server-side it uses the native node.js http module, while on the client (browser) it uses XMLHttpRequests.

To start using Axios in our project, first we need to install it using the command given below.

```
            npm install axios –save
```

Then, we need to include Axios in our component using *import*.

```
            import axios from 'axios';
```

To perform a GET request using axios, we can use axios.get(URL) function which takes one parameter which is the string URL and returns a Promise object.

The .then() block is used to handle promise response and .catch() block is used to handle errors, if any

```
axios.get('URL')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Now we have created a GET request using Axios. What about the response?

The response generated from the server will contain some data. This is the data requested by the user. In order to access this data, we need to use response.data. This data, from the server's response, will be in JSON format.

```
axios.get('URL')

  .then(function (response) {

    console.log(response.data); // This will print response data received from web service.

})
```

To make POST request using axios, we have a method axios.post(url, data) where first parameter is URL string and second parameter is JSON data.

```
axios.post('URL', postData)
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Similar to GET request, response.data is used to get response data received from web service. Its will return response in JSON format.

```
axios.post('URL', data)
  .then(function (response) {
    console.log(response.data); // This will print response data received from web service.
})
```

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Cancelling requests

In some situations, you may no longer care about the result and want to cancel a request that's already sent. This can be done by using a cancel token. The ability to cancel requests was added to Axios in version 1.5 and is based on the **cancelable promises proposal.** Here's a simple example:

```
const source = axios.CancelToken.source();
```

In Axios we need to generate a **Cancel Token.** The token can be generated using the CancelToken.source method.

```
const cancelToken = axios.CancelToken;
const source = cancelToken.source();
```

This **source** is then passed to the axios request:

```
const { data } = await instance.get("/", {
  cancelToken: source.token,
});
```

To check whether a request was cancelled, Axios gives us the "*isCancel*" method, which can be used to determine the cause of the request failing.

```
catch (err) {
 if (axios.isCancel(err)) {
  return "axios request cancelled";
 }
return err;
```

```
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

axios.get('/user/12345', {
  cancelToken: source.token
}).catch(function (thrown) {
  if (axios.isCancel(thrown)) {
    console.log('Request canceled', thrown.message);
  } else {
    // handle error
  }
});

axios.post('/user/12345', {
  name: 'new name'
}, {
  cancelToken: source.token
})

// cancel the request (the message parameter is optional)
source.cancel('Operation canceled by the user.');
```

Handling errors is a very important aspect of any HTTP request as it helps us know whether the request got fulfilled successfully or some error occurred in its process. Also, if some error has occurred, the developer would need to identify that error and resolve it.

We can handle errors by adding the .catch() method to the end of the promise chain.

```
axios.get('URL')

  .then(function (response) {

    console.log(response);

  })

  .catch(function (error) {

    console.log(error.response);

    console.log(error.response.data);

    console.log(error.message)

  });
```

.catch() method will handle an error with the following attributes:

- error.response - It is the error response received from the web service which has details like data, status code and headers.
- error.response.data - It will have response data returned from the web service in JSON format.
- error.message - When there are no responses from the web service, or if something goes wrong while sending a request, we can use this attribute to display a different error message. It can have error string or error status code or both.

To handle errors properly, first we check for a response from web service, if a response is available, we display response error message otherwise we display the message from error.message

```
axios.get('URL')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    if (error.response) {
    console.log(error.response.data.message);
    } else {
    console.log(error.message);
    }
```

6.3 Differences between Axios and Fetch:

Axios	Fetch
Axios has url in request object.	Fetch has no url in request object.
Axios is a stand-alone third party package that can be easily installed.	Fetch is built into most modern browsers; no installation is required as such.
Axios enjoys built-in XSRF protection.	Fetch does not.
Axios uses the data property.	Fetch uses the body property.
Axios' data contains the object.	Fetch's body has to be stringified.
Axios request is ok when status is 200 and statusText is 'OK'.	Fetch request is ok when response object contains the ok property.
Axios 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.
Axios allows cancelling request and request timeout.	Fetch does not.
Axios has the ability to intercept HTTP requests.	Fetch, by default, doesn't provide a way to intercept requests.
Axios has built-in support for download progress.	Fetch does not support upload progress.
Axios has wide browser support.	Fetch only supports Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+ (This is known as Backward Compatibilty).

7. Redux

7.1 Introduction

7.1.1 What is Redux?

Redux is a component based library developed by Dan Abramov and Andrew Clark in 2015 with an inspiration from Facebook's Flux and Elm (functional programming language) for easing the development of react applications. It became popular in the developers' community very soon due to its small size, simplicity and detailed documentation.

Redux is a tool to manage the *data state* and *UI state* of any React application in a much easier manner. Redux also makes the application more consistent. Redux gives us a way to define a global state for our application and maintain it with user's interactions.

7.1.2 Why Redux?

As you know, React flows data in one direction i.e. from parent to child component, communicating between two child components is very difficult and it is not recommended to have direct component to component interaction as shown above as it is more error prone and code becomes very unstructured and difficult to maintain and follow.

Such are the scenarios where the need for Redux arises. Redux helps in easing out the state management by storing all the state's of the application (across all the components) at a single place called as "Store".

For State management

In larger apps with a lot of moving pieces, *state management* becomes a huge concern. **Redux** takes away that effort quite well without performance concerns or trading off testability.

It can be ideally used for developing SPA's where state management becomes complex over time. While Redux was developed keeping React in mind, but it is framework independent and can be used even with angular or jquery applications.

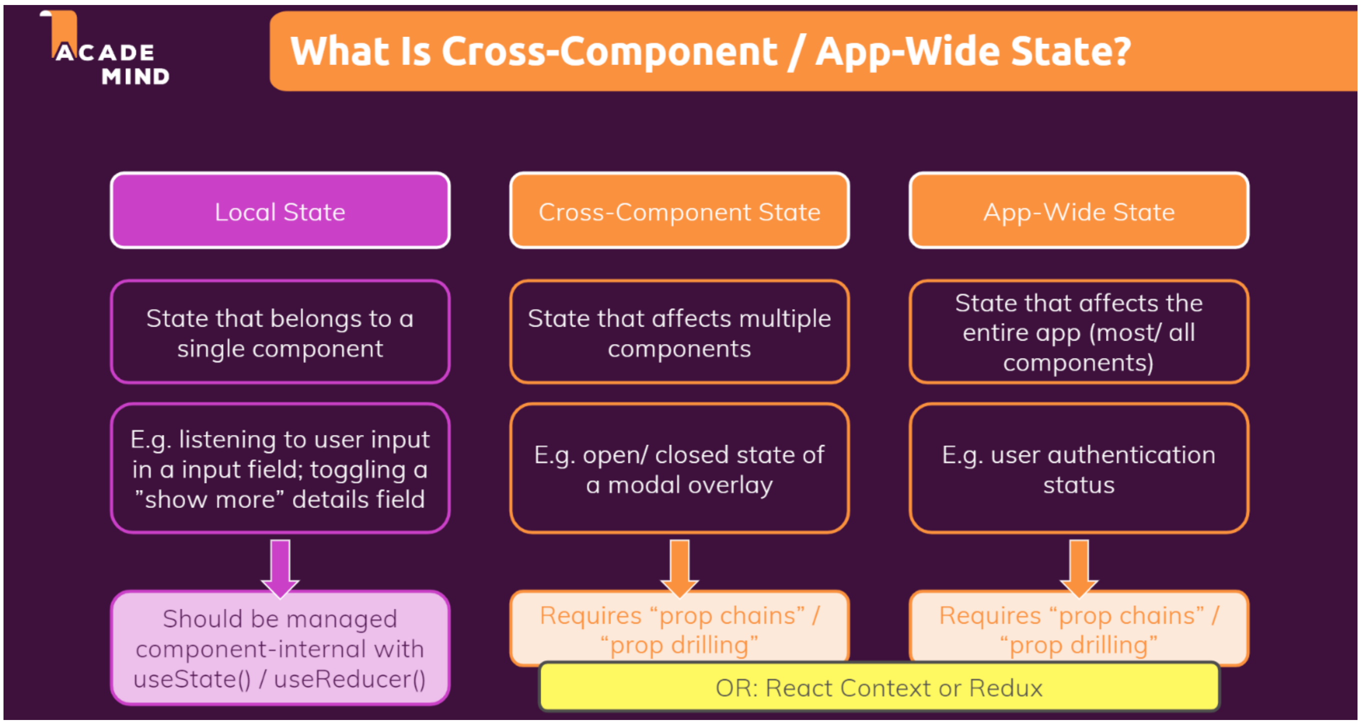
One other reason a lot of developers love Redux is the *developer experience* that comes with it. Redux takes away some of the hassles faced with state management in large applications. It provides us a great developer experience, and makes sure that the ease of testing our app isn’t sacrificed for any of those.

Some of the nice things we get with using Redux include:

- logging
- hot reloading
- time travel
- universal apps
- record and replay—all without doing so much on your end as the developer.

Redux is most useful when in cases when:

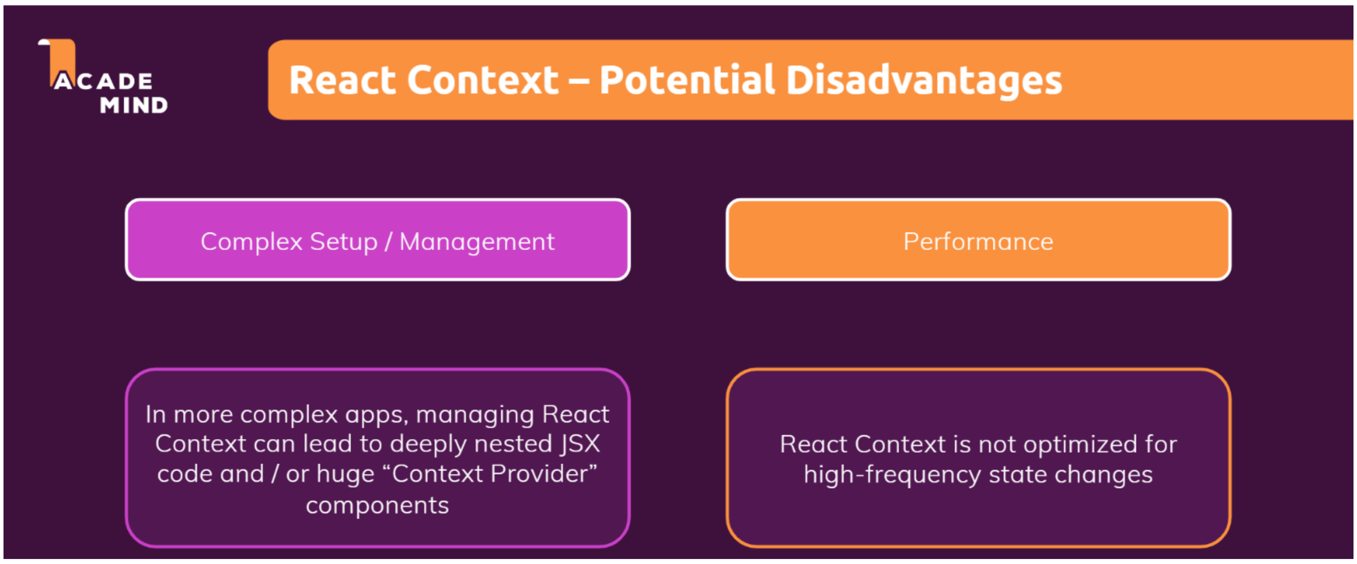
- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people
- You need to see how that state is being updated over time



What Is Cross-Component / App-Wide State?

Local State	Cross-Component State	App-Wide State
State that belongs to a single component	State that affects multiple components	State that affects the entire app (most/ all components)
E.g. listening to user input in a input field; toggling a "show more" details field	E.g. open/ closed state of a modal overlay	E.g. user authentication status
Should be managed component-internal with useState() / useReducer()	Requires "prop chains" / "prop drilling"	Requires "prop chains" / "prop drilling"

OR: React Context or Redux



React Context – Potential Disadvantages

Complex Setup / Management	Performance
In more complex apps, managing React Context can lead to deeply nested JSX code and / or huge "Context Provider" components	React Context is not optimized for high-frequency state changes

7.1.3 Fundamental Principles

Redux has three fundamental principles:

7.1.3.1 Single source of truth

The state of complete application is stored in a **single state object** tree within a store. Having single store in a application makes it easy to create, debug and test. It also means components should read data from this single source and not keep their own version of the same state separately. The state of whole application is centralized and stores in a single state object. In Redux, store is the single source of truth.

7.1.3.2 State is read-only

State is **immutable**. The only way to change a state is by dispatching an action. Since whole application is stored in single state tree, having immutable state make sure that no views or network callbacks wrote directly to state. Instead, they dispatch an action, which have type property which indicates the type of action being performed.

7.1.3.3 Changes are made with pure functions

Pure functions are those functions which do not change input value and always provide same output for same input. In Redux, reducers are pure functions which takes previous state and an action as parameter and return new state. Since reducers are pure functions, it will never change previous state, it will always return new state object.

7.1.4 Main Concepts

The main concepts of Redux architecture are:

- **Store** - Store is the place where all the possible states of all the components of your application are stored. **Store** in Redux can be understood as the **book racks of a library** where the books for all the customers are kept.
- **Actions** - Action is the one which initiates a request to change the 'state'. In our example, **Action** is analogous to **you**. As Action initiates state change, similarly, you are the one who initiates any book borrow or book return from your account.
- **Reducers** - These are the agents that change the state of app, in a smooth and predictable manner. **Reducers** are analogous to the **librarian** who are responsible for any borrow or return of books happening in the library.

7.2 Redux Libraries and Tools

Redux is a small standalone JS library. However, it is commonly used with several other packages:

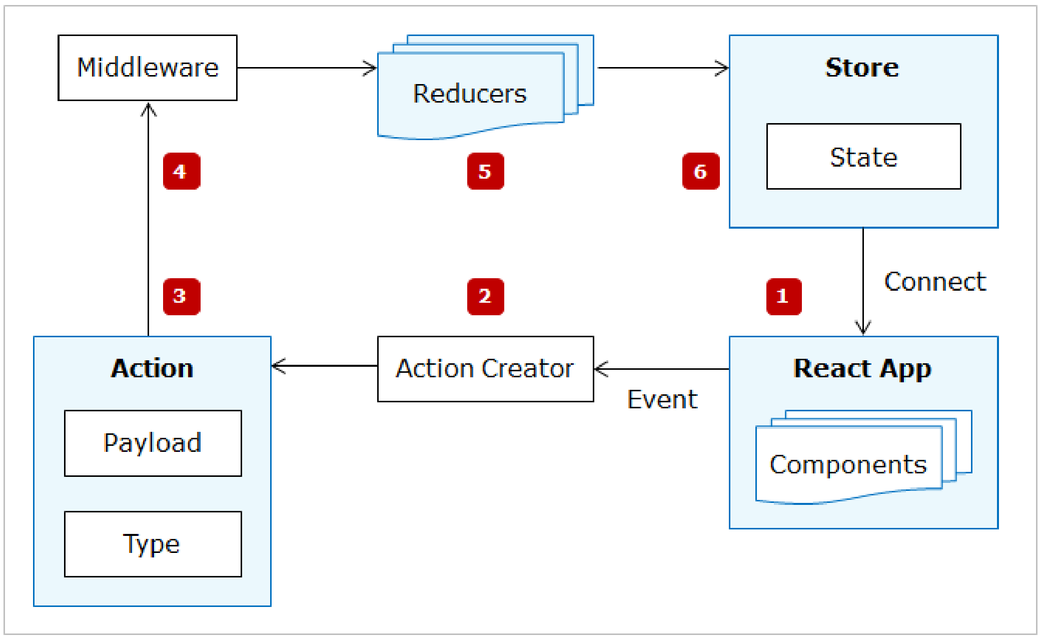
7.2.1 React-Redux

Redux can integrate with any UI framework, and is most frequently used with React. React-Redux is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.

7.2.2 Redux Toolkit

Redux Toolkit is recommended approach for writing Redux logic. It contains packages and functions that are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

7.3 Redux DataFlow



React Components: React components represents the UI rendered on the browser. Components would dispatch actions for events triggered within them and whenever state changes the component renders the current state by connecting to Redux store.

1. **Action Creators:** Action creators are functions which wraps the actual action object.
2. **Action:** Action is a plain JavaScript object. Action is a command to change the state when an event is triggered.
3. **Middleware:** Middleware is a mediator between the action and reducer. Its purpose is to intercept the actions before it reaches the reducer.
4. **Reducer:** Reducer changes the state of the application based on the action triggered. Reducer are functions which accepts action and current state and modify the current state by creating a copy of it based on the action.
5. **Store:** Store is responsible for managing the entire state of the application. State management is centralized in Redux. Action will be dispatched to store using dispatch method of store.

Redux Data Flow explained with Login as an example

Let us observe the Redux data flow by considering Login as an example. Login form is represented by Login component.

1. When user clicks on the login button an action is dispatched.
2. Action reaches action creator which returns an action. Action object contains action type and payload information (if required)
3. Action reaches Reducer which modifies the state
4. If any operation needs to be performed before modifying state then action reaches middleware first and then reaches reducer
5. Reducer updates the modified state to the store
6. React components gets the updated state from the store

7.3.1 Store

The store is the container used to place the entire state of the application.
A store holds the whole state tree of your application. The only way to change the state inside it is to dispatch an action on it.
A Store can be created as shown below

```
import {createStore} from 'redux';
function myReducer() {
}
const store = createStore(myReducer)
```

```
createStore(reducer, [preloadedState], [enhancer])
```

Creates a Redux store that holds the complete state tree of your app. There should only be a single store in your app.

Arguments

- reducer (Function): A reducing function that returns the next state tree, given the current state tree and an action to handle.
- [preloadedState] (any): The initial state. You may optionally specify it to hydrate the state from the server in universal apps, or to restore a previously serialized user session. If you produced reducer with combineReducers, this must be a plain object with the same shape as the keys passed to it. Otherwise, you are free to pass anything that your reducer can understand.
- [enhancer] (Function): The store enhancer. You may optionally specify it to enhance the store with third-party capabilities such as middleware, time travel, persistence, etc. The only store enhancer that ships with Redux is applyMiddleware().

The reducer is passed to the store. Since the state changes are done by reducer the reducer would update the store with the state changes done. Every time reducer changes the state it updates the store and the store would contain the current state.

Redux Store is not a class, rather it is an object consisting of a few methods as given below:

7.3.1.1 getState()

- It does not take any argument and returns the current state tree of your application.
- It is equal to the last value returned by the store's reducers

7.3.1.2 dispatch(action)

- It takes an argument i.e. action. Action is a plain object that describes the changes to made in the application. The action object must have a type field to indicate the type of action being performed. You will see this in detail in redux actions module.
- It dispatches an action for stimulating state changes
- It invokes the store's reducing function with the values of current state (obtained by invoking getState() method) and the action.
- The return value of dispatch(action) is considered as the next state and the change listeners are notified immediately

7.3.1.3 subscribe(listener)

- It is used to add a change listener.
- subscribe() method will be called any time an action is dispatched.
- It takes an argument i.e. listener which is a callback function that gets invoked every time an action is dispatched.
  It returns a function that unsubscribes the change listener.

7.3.1.4 replaceReducer(nextReducer)

- It is called to replace the reducer function currently used by the store to calculate the state by the nextReducer .
- It is an advance API that might be needed in case your app implements code splitting.

In brief, the purpose of redux store is to do following things:

- hold complete application state
- provide access to state by using *store.getState()*
- allow state to be updated by using *dispatch(action)*
- register and de-register listeners by using *subscribe(listener)*

7.3.2 Actions

In Redux since the state management is centralized, the components cannot modify the state directly. Hence Redux dispatches an action whenever there is an event triggered within a component.

An Action is a JavaScript object which describes an event. Action object must contain a type property which describes the type of the action triggered and optionally can have payload information.

One that looks like this:

```
{ type: 'BORROW', qty: 6 }
```

Action object must contain a type property which describes the type of the action triggered and optionally can have payload information.

The payload information can be:

- number
- a boolean value
- a complex object
- or any other value that is serializable to JSON.

The payload information should NOT be a functions or Promises.

7.3.2.1 Action creators

Action creators are the functions that return the action object. We can easily perform an action by invoking action creator.

For example,

```
//can be present in action.js file
//returns an object which contains a type attribute and some payload
export const borrowAction = (quantity) => ({
   type: 'BORROW',
   qty: quantity
});
export const returnAction = (quantity) => ({
   type: 'RETURN',
   qty: quantity
});
```

7.3.3 Reducers

When an action is triggered within any component the state modification is done using Reducers in Redux. Reducers contain the logic for modifying the state in Redux. The state is immutable in Redux, hence state cannot be directly modified by reducers.

Why Immutability?

The benefits of immutability is

- Clarity:

If we are referencing state object in one part of the application and the state object changes in some dramatic way at an unexpected time, not only your application will stop working, but you may have a hard time figuring out what's going wrong. Immutability gives clarity on who changed the value of the state and also it would be easy to track the state updates while debugging.

- Performance:

Consider a state object which has many properties in it. If state is mutable, in order to track the state changes, each and every property has to be tracked by Redux which is an expensive operation. Since state is immutable every time state is modified a new state object is returned. Hence Redux just does a reference comparison. If the old state is not referencing to the same object in memory then it is evident that the state has been modified. Immutability increases performance. React-Redux provides lot of performance optimizations behind the scenes.

- Awesome debugging experience:

Immutability helps in debugging. Time travel debugging is a powerful way to see exactly how your application state is changing over time. We can travel through time as you debug, so we can go back in history and see each specific state change. We can also undo specific state changes and observe how final state changes. We can even turn off individual actions that occurred so you observe what would be the state object if a specific action had not happened. We can play all the interactions back with the click of a button and even select the speed at which it plays back

What is Immutability?

Immutability means objects cannot be changed. To change immutable objects we need to create a copy of it and modify the copy.

In Redux state is an immutable object. State object cannot be changed directly in Redux. Triggering actions are the only way to modify the state. Every time the state is modified a new object should be returned which represents the modifications that are made to state object.

Handling immutability

There are many ways to handle immutability. Object.assign and spread operator for arrays are the most popular approach when working in ES6.

Object.assign()

Consider state object a shown below

```
state={status: 'logged out', value: 'guest',email:'',tel:''}
```

When Login action is triggered we want to change only the status and value properties. We can use Object.assign to create a copy and modify only few properties as shown below.

```
Object.assign({}, state, {
                    status: 'logged in',
                    value: action.value
                })
```

The first parameter to Object.assign is the target which is a new empty object, then we are mixing the new object together with our existing state and also changing the status and value properties. So the result is effectively a clone of our state object but with the state modified after login.

Spread(...) operator

The spread operator is represented using 3 dots (...)

It copies own enumerable properties from a provided object onto a new object.

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };
var clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }
var mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

Reducers in Redux are responsible for copying the current state and modifying the copy and returning the new state.
Reducers are used to achieve immutability in Redux. Reducers are pure functions that are responsible for handling the actions and modify the state of the application based on the action triggered.
Reducers are functions that accept the current state and an action object as arguments and will return the new state.

The initial state of the application has to be passed to the reducer which will be used when we run the application for the first time

Forbidden in Reducers

1. Mutating the arguments passed
2. Performing side effects like API calls, AJAX call, etc
3. Calling other non-pure functions (Ex: date.now(), Math.random())

```
const initialState = {
  todos: [
    { id: 0, text: 'Learn React', completed: true },
    { id: 1, text: 'Learn Redux', completed: false, color: 'purple' },
    { id: 2, text: 'Build something fun!', completed: false, color: 'blue' }
  ],
  filters: {
    status: 'All',
    colors: []
  }
}

// Use the initialState as a default value
export default function appReducer(state = initialState, action) {
  // The reducer normally looks at the action type field to decide what happens
  switch (action.type) {
    // Do something here based on the different types of actions
    default:
      // If this reducer doesn't recognize the action type, or doesn't
      // care about this specific action, return the existing state unchanged
      return state
  }
}
```

7.3.3.1 combineReducers

To achieve modularity we can write multiple reducers in Redux.
Though we split the reducers as our application grows, ultimately we can pass only one reducer object to the createStore method.
We can use the combineReducers() method of Redux to combine multiple reducers into a single reducing function and then pass it to the createStore method.

Whenever an action is triggered the action would reach the root reducer first and then all the reducers get the action from root reducer.
Every reducer will check the action in the switch case. When the action matches the respective reducer will change the state and return the new state to root reducer. Whichever reducer doesn’t find a match will return the current state. Hence we will always have a default case in all the reducers which returns the current state without any modification.
The root reducer will combine the state from all the reducers and create a single new state object and will update the store with the new state.

combineReducers accepts an object where the key names will become the keys in your root state object, and the values are the slice reducer functions that know how to update those slices of the Redux state.

Remember, the key names you give to combineReducers decides what the key names of your state object will be!

```
rootReducer = combineReducers({potato: potatoReducer, tomato: tomatoReducer})
// This would produce the following state object
{
  potato: {
    // ... potatoes, and other state managed by the potatoReducer ...
  },
  tomato: {
    // ... tomatoes, and other state managed by the tomatoReducer, maybe some nice sauce? ...
  }
}
```

7.3.4 Middleware

Middleware need not be a pure function like reducers. It can cause side effects. So whatever functionality we could not put into reducer because reducers are pure can be put in middleware.

A middleware is something that comes in between two main parts of redux - dispatching of an action and the reducer running.

A middleware is a piece of code that can be called on the dispatched action before it reaches the Reducer. Hence, it to intercepts the action before the reducer is invoked.

The Redux docs describe middleware as: "…a third-party extension point between dispatching an action, and the moment it reaches the reducer."

Advantages of middlewares, and what they can do:

Once a middleware intercepts the action, it can carry out a number of operations, including:

- logging information about the store
- making an asynchronous HTTP request
- redirecting the action to another piece of middleware
- dispatching other supplementary actions

Middleware can do any of these before passing the action along to the reducer.

Popular middleware Libraries

- Redux Thunk
- Redux Saga

The most common requirement to use middleware is to support asynchronous actions. Middleware lets us dispatch async actions in addition to our regular actions.

Middleware lets us wrap the dispatch method of the store. The most common requirement to use middleware is to support asynchronous actions. Middleware lets us dispatch async actions in addition to our regular actions.

Redux thunk middleware allows the developer to write action creators which returns a function instead of an action object. The middleware can be used to delay the dispatch of action or dispatch the actions based on certain conditions.

Each middleware receives Store's dispatch and getState functions as named arguments and returns a function.

The function which the action creator returns will receive these two arguments

- dispatch method of store
- getState method of store

The most common async middleware is redux-thunk, which lets you write plain functions that may contain async logic directly.

thunk is available in the redux-thunk library.

```
import thunk from 'redux-thunk';
```

Use applyMiddleware method and pass thunk to it while creating the store.

```
var store = applyMiddleware(thunk)(createStore)(todoApp);
```

or

```
const store = createStore(todoApp, applyMiddleware(thunk))
```

```
export function CallIncrement(step){
  //asynchronous operations can be written here.
  return (dispatch)=>{
    setTimeout(()=>{
        dispatch(onIncrement(step));
    },3000)
  }
}
```

```
export function onIncrement(step){
            return {
            type:"INC",
            step
            }
          }
```

The action creator CallIncrement is dispatching the onIncrement action after 3000 ms. Async operations can be written within the CallIncrement action creator.

7.3.5 Redux Components

7.3.5.1 Presentational components

Presentational components are mainly used to display content on page. So, they consist of markup and styles along with their props. They generally don't deal with data fetching and state updates. Also, they do not have their own state.

```
import React, { Component } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
// a presentational component which has no state of its own
export default class AppRedux extends Component {
    userInputQty = 0;
    render() {
        return (<div>
        <div className={'col-5 offset-1 card'}>
            <br/>
            <table>
                <tbody>
                    <tr>
                        <th><h3>Books in library are: </h3></th>
                        <td><h3>{+this.props.pBookBalance }</h3></td>
                    </tr>
                    <tr>
                        <th>Enter the amount for transaction: </th>
                        <td><input onChange={(e)=>{ this.userInputQty = e.target.value;}}
type='number' className={'form-control'}/></td>
                    </tr>
                    <tr>
                        <td>
                            <button onClick={()=>{this.props.debit(this.userInputQty)}} className={'btn
btn-primary'}>
                                Borrow </button>
                        </td>
                        <td>
                            <button onClick={()=>{this.props.credit(this.userInputQty)}} className={'btn
btn-success'}>
                                Debit </button>
                        </td>
                    </tr>
                </tbody>
            </table>
            <br/>
        </div>
        </div>);
    }
}
```

7.3.5.2 Container components

Container components are responsible for connecting to the Redux store and fetching the updated state from Redux **store**. Once container components get the state from Redux store they pass the updated state to presentational components as props.
Container components can be written by developer by subscribing to the store using **store.subscribe()** and the current state can be fetched from the store using **store.getState()** method.
But as already discussed React Redux has already implemented the necessary performance optimizations. Hence we generate container components using the methods provided by react-redux library, rather than writing the container components by hand.
container component can be generated by connect() function of react-redux library. To use connect() we need to define two functions:

- **mapStateToProps** - This function helps in transforming the current state present in the redux store as props and then the same is passed to it's corresponding presentational component.

- **mapDispatchToProps** - This function is used by container components to dispatch actions.

Let us have a look at the container for app component i.e. AppContainer.

```
//can be present in AppContainer.js file
import { borrowAction , returnAction } from './redux';
import AppRedux  from './AppRedux'; //presentational component that contains the view
import { connect } from 'react-redux';
const mapStatetoProps = state => ({
   pBookBalance : state.accountBooks
});
const mapDispatchToProps = {
   borrowBooks : borrowAction ,
   returnBooks : returnAction
}
export default connect(mapStatetoProps,mapDispatchToProps)(AppRedux);
```

useDispatch()

```
const dispatch = useDispatch()
```

This hook returns a reference to the dispatch function from the Redux store. You may use it to dispatch actions as needed.

useSelector()

```
const result: any = useSelector(selector: Function, equalityFn?: Function)
```

Allows you to extract data from the Redux store state, using a selector function.
The selector is approximately equivalent to the **mapStateToProps** argument to connect conceptually.
The selector will be called with the entire Redux store state as its only argument.
The selector will be run whenever the function component renders (unless its reference hasn't changed since a previous render of the component so that a cached result can be returned by the hook without re-running the selector).

useSelector() will also subscribe to the Redux store, and run your selector whenever an action is dispatched.

However, there are some differences between the selectors passed to **useSelector()** and a **mapState** function:

- The selector may return any value as a result, not just an object. The return value of the selector will be used as the return value of the useSelector() hook.
- When an action is dispatched, useSelector() will do a reference comparison of the previous selector result value and the current result value. If they are different, the component will be forced to re-render. If they are the same, the component will not re-render.
- The selector function does not receive an ownProps argument. However, props can be used through closure (see the examples below) or by using a curried selector.
- Extra care must be taken when using memoizing selectors (see examples below for more details).
- useSelector() uses strict === reference equality checks by default, not shallow equality (see the following section for more details).

When the function component renders, the provided selector function will be called and its result will be returned from the useSelector() hook. (A cached result may be returned by the hook without re-running the selector if it's the same function reference as on a previous render of the component.)

However, when an action is dispatched to the Redux store, useSelector() only forces a re-render if the selector result appears to be different than the last result. As of v7.1.0-alpha.5, the default comparison is a strict === reference comparison. This is different than connect(), which uses shallow equality checks on the results of mapState calls to determine if re-rendering is needed. This has several implications on how you should use useSelector().

With mapState, all individual fields were returned in a combined object. It didn't matter if the return object was a new reference or not - connect() just compared the individual fields. With useSelector(), returning a new object every time will always force a re-render by default. If you want to retrieve multiple values from the store, you can:

- Call useSelector() multiple times, with each call returning a single field value
- Use Reselect or a similar library to create a memoized selector that returns multiple values in one object, but only returns a new object when one of the values has changed.
- Use the shallowEqual function from React-Redux as the equalityFn argument to useSelector(), like:

```
import { shallowEqual, useSelector } from 'react-redux'

// later
const selectedData = useSelector(selectorReturningObject, shallowEqual)
```

The <Provider> component makes the Redux store available to any nested components that need to access the Redux store.

Since any React component in a React Redux app can be connected to the store, most applications will render a <Provider> at the top level, with the entire app's component tree inside of it.
The Hooks and connect APIs can then access the provided store instance via React's Context mechanism.

When rendering the root component to the DOM, the root component should be wrapped within the <Provider> component as shown below.

```
//other imports if any
import React from 'react';
import ReactDOM from 'react-dom';
import AppRedux from './AppContainer';
import { Provider } from 'react-redux';
import { bankStore } from './redux';
ReactDOM.render(
  <Provider store={ bankStore }>
    <AppRedux />
  </Provider>,
  document.getElementById("root")
);
```

Demo:

Store

```
import { createStore } from "redux";

const initialState = { counter: 0, showCounter: true };
const counterReducer = (state = initialState, action) => {
  if (action.type === "INC") {
    return {
      counter: state.counter + 1,
      showCounter: state.showCounter,
    };
  }
  if (action.type === "DEC") {
    return {
      counter: state.counter - 1,
      showCounter: state.showCounter,
    };
  }
  if (action.type === "INCV") {
    return {
      counter: state.counter + action.amount,
      showCounter: state.showCounter,
    };
  }
  if (action.type === "toogle") {
    return {
      showCounter: !state.showCounter,
      counter: state.counter,
    };
  }
  return state;
};
const store = createStore(counterReducer);

export default store;
```

Provider

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

import './index.css';
import App from './App';
import store from './store/index';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root') );
```

Counter

```
import classes from "./Counter.module.css";
import { useSelector, useDispatch, connect } from "react-redux";
import React from "react";
import { Component } from "react";
const Counter = () => {
  const dispatch = useDispatch();
  const counter = useSelector((state) => state.counter);
  const show = useSelector((state) => state.showCounter);
  const toggleCounterHandler = () => {
    dispatch({ type: "toogle" });
  };

  const IncrementHandler = () => {
    dispatch({ type: "INC" });
  };

  const DecrementHandler = () => {
    dispatch({ type: "DEC" });
  };

  const increaseHandler = () => {
    dispatch({ type: "INCV", amount: 5 });
  };
  return (
    <main className={classes.counter}>
      <h1>Redux Counter</h1>
      {show && <div className={classes.value}>{counter}</div>}
      <button onClick={toggleCounterHandler}>Toggle Counter</button>
      <div>
        <button onClick={IncrementHandler}>Increment</button>
        <button onClick={increaseHandler}>Increase by 5</button>
        <button onClick={DecrementHandler}>Decrement</button>
      </div>
    </main>
  );
};
export default Counter;

// class Counter extends Component {
//   toggleCounterHandler() {}

//   IncrementHandler() {
//     this.props.increment();
//   }

//   DecrementHandler() {
//     this.props.decrement();
//   }
//   render() {
//     return (
//       <main className={classes.counter}>
//        <h1>Redux Counter</h1>
//        <div className={classes.value}>{this.props.counter}</div>
//        <button onClick={this.toggleCounterHandler}>Toggle Counter</button>
//        <div>
//          <button onClick={this.IncrementHandler.bind(this)}>Increment</button>
//          <button onClick={this.DecrementHandler.bind(this)}>Decrement</button>
//        </div>
//      </main>
//    );
//   }
// }
// const mapStateToProps = (state) => {
//   return {
//     counter: state.counter,
//   };
// };

// const mapDispatchToProps = (dispatch) => {
//   return {
//     increment: () => {
//       dispatch({ type: "INC" });
//     },
//     decrement: () => {
//       dispatch({ type: "DEC" });
//     },
//   };
// };
// export default connect(mapStateToProps, mapDispatchToProps)(Counter);
```

7.4 Redux-ToolKit

The Redux Toolkit package is intended to be the standard way to write Redux logic. It was originally created to help address three common concerns about Redux:

- "Configuring a Redux store is too complicated"
- "I have to add a lot of packages to get Redux to do anything useful"
- "Redux requires too much boilerplate code"

Installation

Using Create React App

The recommended way to start new apps with React and Redux is by using the official Redux+JS template or Redux+TS template for Create React App, which takes advantage of Redux Toolkit and React Redux's integration with React components.

```
# Redux + Plain JS template

npx create-react-app my-app --template redux


# Redux + TypeScript template

npx create-react-app my-app --template redux-typescript
```

Redux Toolkit is available as a package on NPM for use with a module bundler or in a Node application:

NPM

```
npm install @reduxjs/toolkit
```

7.4.1 createSlice

Redux Toolkit includes a createSlice function that will auto-generate the action types and action creators for you, based on the names of the reducer functions you provide.

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.

Redux requires that we write all state updates immutably, by making copies of data and updating the copies. However, Redux Toolkit's createSlice and createReducer APIs use Immer inside to allow us to write "mutating" update logic that becomes correct immutable updates

.

```
 const postsSlice = createSlice({
  name: 'posts',
  initialState: [],
  reducers: {
    createPost(state, action) {},
    updatePost(state, action) {},
    deletePost(state, action) {},
  },
})

console.log(postsSlice)
/*
{
    name: 'posts',
    actions : {
        createPost,
        updatePost,
        deletePost,
    },
    reducer
}
*/

const { createPost } = postsSlice.actions

console.log(createPost({ id: 123, title: 'Hello World' }))
// {type : "posts/createPost", payload : {id : 123, title : "Hello World"}}
```

createSlice looked at all of the functions that were defined in the reducers field, and for every "case reducer" function provided, generates an action creator that uses the name of the reducer as the action type itself.
So, the createPost reducer became an action type of "posts/createPost", and the createPost() action creator will return an action with that type.
Most of the time, you'll want to define a slice, and export its action creators and reducers. The recommended way to do this is using ES6 destructuring and export syntax:

```
const postsSlice = createSlice({
  name: 'posts',
  initialState: [],
  reducers: {
    createPost(state, action) {},
    updatePost(state, action) {},
    deletePost(state, action) {},
  },
})

// Extract the action creators object and the reducer
const { actions, reducer } = postsSlice
// Extract and export each action creator by name
export const { createPost, updatePost, deletePost } = actions
// Export the reducer, either as a default or named export
export default reducer
```

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

7.4.2 configureStore

A friendly abstraction over the standard Redux createStore function that adds good defaults to the store setup for a better development experience.

Parameters

configureStore accepts a single configuration object parameter, with the following options:

```
type ConfigureEnhancersCallback = (
  defaultEnhancers: StoreEnhancer[]
) => StoreEnhancer[]

interface ConfigureStoreOptions<
  S = any,
  A extends Action = AnyAction,
  M extends Middlewares<S> = Middlewares<S>
> {
  /**
   * A single reducer function that will be used as the root reducer, or an
   * object of slice reducers that will be passed to `combineReducers()`.
   */
  reducer: Reducer<S, A> | ReducersMapObject<S, A>

  /**
   * An array of Redux middleware to install. If not supplied, defaults to
   * the set of middleware returned by `getDefaultMiddleware()`.
   */
  middleware?: ((getDefaultMiddleware: CurriedGetDefaultMiddleware<S>) => M) | M

  /**
   * Whether to enable Redux DevTools integration. Defaults to `true`.
   *
   * Additional configuration can be done by passing Redux DevTools options
   */
  devTools?: boolean | DevToolsOptions

  /**
   * The initial state, same as Redux's createStore.
   * You may optionally specify it to hydrate the state
   * from the server in universal apps, or to restore a previously serialized
   * user session. If you use `combineReducers()` to produce the root reducer
   * function (either directly or indirectly by passing an object as `reducer`),
   * this must be an object with the same shape as the reducer map keys.
   */
  preloadedState?: DeepPartial<S extends any ? S : S>

  /**
   * The store enhancers to apply. See Redux's `createStore()`.
   * All enhancers will be included before the DevTools Extension enhancer.
   * If you need to customize the order of enhancers, supply a callback
   * function that will receive the original array (ie, `[applyMiddleware]`),
   * and should return a new array (such as `[applyMiddleware, offline]`).
   * If you only need to add middleware, you can use the `middleware` parameter instead.
   */
  enhancers?: StoreEnhancer[] | ConfigureEnhancersCallback
}

function configureStore<S = any, A extends Action = AnyAction>(
  options: ConfigureStoreOptions<S, A>
): EnhancedStore<S, A>
```

The simplest way to use it is to just pass the root reducer function as a parameter named reducer:

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducers'
const store = configureStore({
  reducer: rootReducer,
})
export default store
```

You can also pass an object full of "slice reducers", and configureStore will call combineReducers for you:

```
import { configureStore } from '@reduxjs/toolkit'
import usersReducer from './usersReducer'
import postsReducer from './postsReducer'
const store = configureStore({
  reducer: {
    users: usersReducer,
    posts: postsReducer,
  },

})
export default store
```

Note that this only works for one level of reducers. If you want to nest reducers, you'll need to call combineReducers yourself to handle the nesting.

- The basic Redux createStore function takes positional arguments: (rootReducer, preloadedState, enhancer). Sometimes it's easy to forget which parameter is which.
- The process of setting up middleware and enhancers can be confusing, especially if you're trying to add several pieces of configuration.
- The Redux DevTools Extension docs initially suggest using some hand-written code that checks the global namespace to see if the extension is available. Many users copy and paste those snippets, which make the setup code harder to read.

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export default configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

Use Redux State and Actions in React Components

Now we can use the React-Redux hooks to let React components interact with the Redux store. We can read data from the store with useSelector, and dispatch actions using useDispatch.

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  )}
```

Demo:

Store

```
import { createSlice, configureStore } from "@reduxjs/toolkit";
const initialCounterState = { counter: 0, showCounter: true };

const counterSlice = createSlice({
  name: "counter",
  initialState: initialCounterState,
  reducers: {
    increment(state) {
      state.counter++;
    },
    decrement(state) {
      state.counter--;
    },
    increase(state, action) {
      state.counter += action.payload;
    },
    toogleCounter(state) {
      state.showCounter = !state.showCounter;
    },
  },
});

const initialAuthState = {
  isAuthenticated: false,
};

const AuthSlice = createSlice({
  name: "Auth",
  initialState: initialAuthState,
  reducers: {
    login(state) {
      state.isAuthenticated = true;
    },
    logout(state) {
      state.isAuthenticated = false;
    },
  },
});
const store = configureStore({
  reducer: { counter: counterSlice.reducer, auth: AuthSlice.reducer },
});

export const counterActions = counterSlice.actions;
export const authActions = AuthSlice.actions;
export default store;
```

Counter

```
import classes from "./Counter.module.css";
import { useSelector, useDispatch } from "react-redux";
import React from "react";
import { counterActions } from "../Store/index";

const Counter = () => {
  const dispatch = useDispatch();
  const counter = useSelector((state) => state.counter.counter);
  const show = useSelector((state) => state.counter.showCounter);
  const toggleCounterHandler = () => {
    dispatch(counterActions.toogleCounter());
  };

  const IncrementHandler = () => {
    dispatch(counterActions.increment());
  };

  const DecrementHandler = () => {
    dispatch(counterActions.decrement());
  };

  const increaseHandler = () => {
    dispatch(counterActions.increase(5));
  };
  return (
    <main className={classes.counter}>
      <h1>Redux Counter</h1>
      {show && <div className={classes.value}>{counter}</div>}
      <button onClick={toggleCounterHandler}>Toggle Counter</button>
      <div>
        <button onClick={IncrementHandler}>Increment</button>
        <button onClick={increaseHandler}>Increase by 5</button>
        <button onClick={DecrementHandler}>Decrement</button>
      </div>
    </main>
  );
};

export default Counter;
```

Auth

```
import classes from "./Auth.module.css";
import { authActions } from "../Store/index";
import { useDispatch } from "react-redux";

const Auth = () => {
  const dispatch = useDispatch();
  const loginHandler = (e) => {
    e.preventDefault();
    dispatch(authActions.login());
  };
  return (
    <main className={classes.auth}>
      <section>
        <form onSubmit={loginHandler}>
          <div className={classes.control}>
            <label htmlFor="email">Email</label>
            <input type="email" id="email" />
          </div>
          <div className={classes.control}>
            <label htmlFor="password">Password</label>
            <input type="password" id="password" />
          </div>
          <button>Login</button>
        </form>
      </section>
    </main>
  );
};

export default Auth;
```

Provider

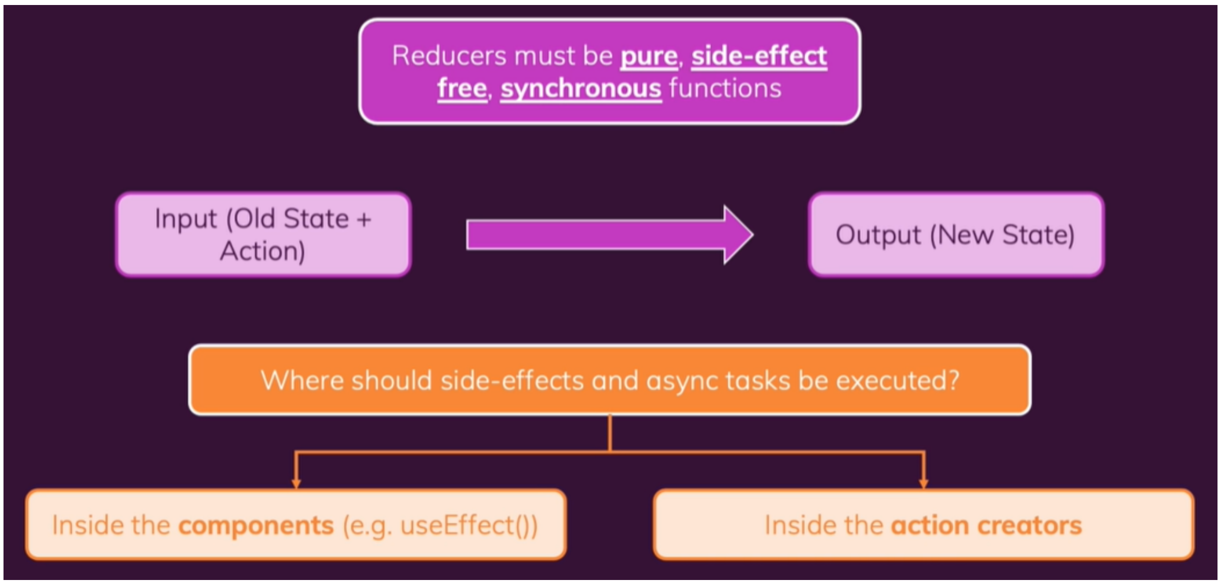
```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./Store/index";
import "./index.css";
import App from "./App";
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

7.4.3 Async Actions

The most common requirement to use middleware is to support asynchronous actions. Middleware lets us dispatch async actions in addition to our regular actions.

Middleware lets us wrap the dispatch method of the store. The most common requirement to use middleware is to support asynchronous actions. Middleware lets us dispatch async actions in addition to our regular actions.

The most common async middleware is redux-thunk, which lets you write plain functions that may contain async logic directly. Redux Toolkit's configureStore function automatically sets up the thunk middleware by default.


Reducers must be pure, side-effect free, synchronous functions
Input (Old State + Action)
Output (New State)
Where should side-effects and async tasks be executed?
Inside the components (e.g. useEffect())
Inside the action creators

Inside the component(eg. useEffect())

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

const uiSlice = createSlice({
  name: 'ui',
  initialState: { cartIsVisible: false, notification: null },
  reducers: {
    toggle(state) {
      state.cartIsVisible = !state.cartIsVisible;
    },
    showNotification(state, action) {
      state.notification = {
        status: action.payload.status,
        title: action.payload.title,
        message: action.payload.message,
      };
    },
  },
});

export const uiActions = uiSlice.actions;
```

```
import { Fragment, useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';

import Cart from './components/Cart/Cart';
import Layout from './components/Layout/Layout';
import Products from './components/Shop/Products';
import { uiActions } from './store/ui-slice';
import Notification from './components/UI/Notification';

let isInitial = true;

function App() {
  const dispatch = useDispatch();
  const showCart = useSelector((state) => state.ui.cartIsVisible);
  const cart = useSelector((state) => state.cart);
  const notification = useSelector((state) => state.ui.notification);

  useEffect(() => {
    const sendCartData = async () => {
      dispatch(
        uiActions.showNotification({
          status: 'pending',
          title: 'Sending...',
          message: 'Sending cart data!',
        })
      );
      const response = await fetch(
        'https://react-http-6b4a6.firebaseio.com/cart.json',
        {
```

```
        method: 'PUT',
        body: JSON.stringify(cart),
      }
    );

    if (!response.ok) {
      throw new Error('Sending cart data failed.');
    }

    dispatch(
      uiActions.showNotification({
        status: 'success',
        title: 'Success!',
        message: 'Sent cart data successfully!',
      })
    );
  };

  if (isInitial) {
    isInitial = false;
    return;
  }

  sendCartData().catch((error) => {
    dispatch(
      uiActions.showNotification({
        status: 'error',
        title: 'Error!',
        message: 'Sending cart data failed!',
      })
    );
  });
 }, [cart, dispatch]);

 return (
  <Fragment>
    {notification && (
      <Notification
        status={notification.status}
        title={notification.title}
        message={notification.message}
      />
    )}
    <Layout>
      {showCart && <Cart />}
      <Products />
    </Layout>
  </Fragment>
 );
}

export default App
```

Inside action creators(thunk)

```
import { createSlice } from "@reduxjs/toolkit";
import { uiActions } from "./ui-slice";
const cartSlice = createSlice({
  name: "cart",
  initialState: {
    items: [],
    totalQuantity: 0,
  },
  reducers: {
    addItemtoCart(state, action) {
      const newItem = action.payload;
      const existingItem = state.items.find((item) => item.id === newItem.id);

      if (!existingItem) {
        state.items.push({
          id: newItem.id,
          name: newItem.title,
          price: newItem.price,
          quantity: 1,
          totalPrice: newItem.price,
        });
        state.totalQuantity++;
      } else {
        existingItem.quantity++;
        existingItem.totalPrice += newItem.price;
        state.totalQuantity++;
      }
    },
    removeItemFromCart(state, action) {
      const id = action.payload;
      const existingItem = state.items.find((item) => item.id === id);
      state.totalQuantity--;
      if (existingItem.quantity === 1) {
        state.items = state.items.filter((item) => item.id !== id);
      } else {
        existingItem.quantity--;

        existingItem.totalPrice -= existingItem.price;
      }
    },
  },
});

export const sendCartData = (cart) => {
  return async (dispatch) => {
    dispatch(
      uiActions.showNotifications({
        status: "pending",
        title: "sending",
        message: "Sending Cart data",
      })
    );

    const sendRequest = async () => {
      const response = await fetch(
        "https://react-httppost-58957-default-rtdb.firebaseio.com/cart.json",
        {
          method: "put",
          body: JSON.stringify(cart),
        }
      );
      if (!response.ok) {
        throw new Error("Sending Cart data failed");
      }
    };
    try {
      await sendRequest();
      dispatch(
        uiActions.showNotifications({
          status: "success",
          title: "Done",
          message: "Sent Cart data successfully",
        })
      );
    } catch (error) {
      dispatch(
        uiActions.showNotifications({
          status: "error",
          title: "Failed",
          message: "Sent Cart data failed",
        })
      );
    }
  };
};
export default cartSlice;
export const cartActions = cartSlice.actions;
```

```
import { useEffect, Fragment } from "react";
import { useDispatch, useSelector } from "react-redux";
import Cart from "./components/Cart/Cart";
import Layout from "./components/Layout/Layout";
import Products from "./components/Shop/Products";
import Notification from "./components/UI/Notification";
import { sendCartData } from "./Store/cart-slice";

let isInitial = true;
function App() {
  const dispatch = useDispatch();
  const cartIsVisible = useSelector((state) => state.ui.cartIsVisible);

  const cart = useSelector((state) => state.cart);
  const notificationStatus = useSelector((state) => state.ui.notification);

  useEffect(() => {
    if (isInitial) {
      isInitial = false;
      return;
    }
    dispatch(sendCartData(cart));
  }, [cart, dispatch]);
  return (
    <Fragment>
      {notificationStatus && (
        <Notification
          status={notificationStatus.status}
          title={notificationStatus.title}
          message={notificationStatus.message}
        />
      )}
      <Layout>
        {cartIsVisible && <Cart />}
        <Products />
      </Layout>
    </Fragment>
  );
}

export default App;
```

8. Routing

Routing is a process in which a user is directed to different pages based on their action or request.

ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

In React, we will observe that *routing take place as the app renders*.

8.1 React Router

React contains three different packages for routing. These are:

- react-router: It provides the core routing components and functions for the React Router applications.
- react-router-native: It is used for mobile applications.
- react-router-dom: It is used for web applications design.

8.1.1 Installation

Install react-router-dom library by running the npm command in your application folder:

```
npm install react-router-dom
```

8.1.2 <BrowserRouter />

The first new component we use is called <BrowserRouter>.

It is a primary component containing the entire routing configuration. It is a more popular one because it uses the HTML5 History API to keep track of your router history.

It is used for handling the dynamic URL.

```
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";
import "./index.css";
import App from "./App";

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById("root")
);
```

As of now, the App component has the BrowserRouter directly present inside the render() method.

This means that the App component will be loaded in the root element of index.html and within this App component, the first component to be loaded will be the BrowserRouter which can handle incoming dynamic requests for various other child components like 'Home' or 'NewsFeed'.

Best Practices: It is considered a good practice to code the router logic in the App component itself. It will initialize the router as soon component is loaded in the browser

8.1.3 <Link />

The <Link> element provides declarative, accessible navigation around our application. <Link/> is similar to an anchor <a> tag and helps us navigate to a different url.

The <Link> element has a prop called 'to' , which help us specify the the new url that we want to navigate to.

But on clicking the link, only the URL changes. The corresponding component is not rendered yet.

```
<Link to={'/contact'}> Contact </Link>
```

to: object

An object that can have any of the following properties:

- pathname: A string representing the path to link to.
- search: A string representation of query parameters.
- hash: A hash to put in the URL, e.g. #a-hash.
- state: State to persist to the location.

8.1.4 <NavLink/>

A special version of the <Link> that will add styling attributes to the rendered element when it matches the current URL.

NavLink component provides 2 props to style the active link, they are:

8.1.4.1 activeClassName

Using activeClassName prop we can bind the CSS class to it as follows:

```
<NavLink to="/" className="nav-item nav-link" exact activeClassName="active">Home</NavLink>
<NavLink  to="/about" className="nav-item nav-link" activeClassName="active" >About Us</NavLink>
```

Add the below styles in CSS file:

```
.active{

background-color: black;
}
```

8.1.4.2 activeStyle

Using activeStyle prop, inline styles can be defined to style the link as follows:

```
<NavLink to="/" className="nav-item nav-link" exact activeStyle={{
    background:'white',
    color:'black'
    }}>
    Home
</NavLink>
```

8.1.5 <Route/>

This tag is used to specify which component to render when the user clicks on a link and the URL is updated.
The Route component needs us to specify two props - path and component.
The path specifies the URL and component specifies which component should be loaded dynamically.
So we can say the <Route> tag is responsible for displaying the appropriate component after the URL is changed by clicking a link.

There are 3 ways to specify the component to be rendered

<Route component>

A React component to render only when the location matches. It will be rendered with route props.

```
<Route path="/user" component={User} />
```

When you use component (instead of render or children, below) the router uses React.createElement to create a new React element from the given component.
That means if you provide an inline function to the component prop, you would create a new component on every render. This results in the existing component unmounting and the new component mounting instead of just updating the existing component. When using an inline function for inline rendering, use the render or the children prop.

<Route render>

```
<Route path="/home" render={() => <div>Home</div>} />
```

This allows for convenient inline rendering and wrapping without the undesired remounting explained above.

Instead of having a new React element created for you using the component prop, you can pass in a function to be called when the location matches. The render prop function has access to all the same route props (match, location and history) as the component render prop.

<Route children> function

```
<Route path="user/:id" children={<UserComponent/>} />
```

Sometimes you need to render whether the path matches the location or not.

In these cases, you can use the function children prop.

It works exactly like render except that it gets called whether there is a match or not.

The children render prop receives all the same route props as the component and render methods, except when a route fails to match the URL, then match is null. This allows you to dynamically adjust your UI based on whether or not the route matches.

This could also be useful for animations.

exact: This property tells Route to match the exact path.

Warning: <Route children> takes precedence over both <Route component> and <Route render> so don't use more than one in the same <Route>.

All three render methods will be passed the same three route props

Match

A match object contains information about how a <Route path> matched the URL. match objects contain the following properties:

- params - (object) Key/value pairs parsed from the URL corresponding to the dynamic segments of the path
- isExact - (boolean) true if the entire URL was matched (no trailing characters)
- path - (string) The path pattern used to match. Useful for building nested <Route>s
- url - (string) The matched portion of the URL. Useful for building nested <Link>s

You'll have access to match objects in various places:

```
Route component as this.props.match
Route render as ({ match }) => ()
Route children as ({ match }) => ()
```

If a Route does not have a path, and therefore always matches, you'll get the closest parent match.

Location

Locations represent where the app is now, where you want it to go, or even where it was. It even where it was.

It looks like this:

```
{
key: 'ac3df4', // not with HashHistory!
  pathname: '/somewhere',
  search: '?some=search-string',
  hash: '#howdy',
  state: {
    [userDefined]: true
  }
}
```

The router will provide you with a location object in a few places:

```
Route component as this.props.location
Route render as ({ location }) => ()
Route children as ({ location }) => ()
```

It is also found on history.location but you shouldn't use that because it's mutable. You can read more about that in the history doc.

You can provide locations instead of strings to the various places that navigate:

- Web Link to
- Native Link to
- Redirect to
- history.push
- history.replace

Normally you just use a string, but if you need to add some "location state" that will be available whenever the app returns to that specific location, you can use a location object instead. This is useful if you want to branch UI based on navigation history instead of just paths (like modals).

history

The term "history" and "history object" in this documentation refers to the history package, which is one of only 2 major dependencies of React Router (besides React itself), and which provides several different implementations for managing session history in JavaScript in various environments.

The following terms are also used:

- "browser history" - A DOM-specific implementation, useful in web browsers that support the HTML5 history API
- "hash history" - A DOM-specific implementation for legacy web browsers
- "memory history" - An in-memory history implementation, useful in testing and non-DOM environments like React Native

history objects typically have the following properties and methods:

- length - (number) The number of entries in the history stack
- action - (string) The current action (PUSH, REPLACE, or POP)
- location - (object) The current location. May have the following properties:
- pathname - (string) The path of the URL
- search - (string) The URL query string
- hash - (string) The URL hash fragment
- state - (object) location-specific state that was provided to e.g. push(path, state) when this location was pushed onto the stack. Only available in browser and memory history.
- push(path, [state]) - (function) Pushes a new entry onto the history stack
- replace(path, [state]) - (function) Replaces the current entry on the history stack
- go(n) - (function) Moves the pointer in the history stack by n entries
- goBack() - (function) Equivalent to go(-1)
- goForward() - (function) Equivalent to go(1)
- block(prompt) - (function) Prevents navigation (see the history docs)

history is mutable

The history object is mutable. Therefore it is recommended to access the location from the render props of <Route>, not from history.location. This ensures your assumptions about React are correct in lifecycle hooks

8.1.5.1 Parameterized Routes

Sometimes, we need to pass certain data or parameters along with the URL, such that the corresponding component can be rendered. These parameters are called **route parameters**

To render a component, based on the parameterized URL changes the route can be configured as shown:

```
<Route path="/profile/:username" component={UserProfile}/>
```

To an end user, this URL will not look any different from a non-parameterized URL.

But in the code, the " : " in front of username signifies the dynamic parameter.

Using the parameter passed in the URL.

In the above code snippet, **UserProfile** component gets rendered, only when a parameter is passed to the path.

Here, ***username*** is the route param. It is like we have declared a variable to store the parameter provided by the user. It makes sense to now use the passed parameter in the component we want to render, i.e, *UserProfile* component.

In *UserProfile* component(class), the route param value can be accessed as follows:

```
this.props.match.params.username
```

match: react-router-dom passes in a prop called match into every route that is rendered. Inside this match object there is an other object called params.

params: It is an object containing the passed URL parameters

username: In this example, the parameter is called *username*. The keyword has to match the keyword used in the route path.

In *UserProfile* component(functional), the route param value can be accessed as follows:

useParams() returns an object of key/value pairs of URL parameters. Use it to access match.params of the current <Route>.

```
const params=useParams()

console.log(params.username)
```

8.1.5.2 Handling 404 case in React application

While configuring the Routes for an application, we will have a common Route, which will be rendered if none of the other Routes are matched.

For example, When a user try to access nonexistent pages in an application or when the user mistypes a URL, we can render a 404 error page/view, by rendering a common Route that has no path configured for it. And we can handle 404 cases by redirecting to any particular Route as well.

Let us understand how to create a Route for handling 404 cases.

We can write the below Route as that last one, in the list of Routes configured:

```
<Route component={NotFound}>
```

This can also be written as follows:

```
<Route path='*' component={NotFound}>
```

When none of the routes are matched or handled prior to it, then the NotFound component will be rendered.

8.1.5.3 Configuring Routes Programmatically

When we want to navigate from one component to another component, we can use history prop.

Components that are passed to the <Route/> component will get history, location, and match as props.

So when we want to navigate from one component (class) to another component, we can achieve it as shown below

```
this.props.history.push('url')
```

The **useHistory** hook gives you access to the history instance that you may use to navigate.

```
import { useHistory } from "react-router-dom";

function HomeButton() {
  let history = useHistory();

  function handleClick() {
    history.push("/home");
  }

  return (
    <button type="button" onClick={handleClick}>
      Go home
    </button>
  );
}
```

8.1.5.4 Query Parameters

A query parameter is used to filter down a resource.

A typical example is using parameters like pageSize or page to indicate to the backend that you only want a small slice of content and not the full list which can be millions of rows potentially.

Query parameters are found after the ? character in the url which would make the url look like this /products?page=1&pageSize=20.

useLocation

The useLocation hook returns the location object that represents the current URL.

You can think about it like a useState that returns a new location whenever the URL changes.

This could be really useful e.g. in a situation where you would like to trigger a new “page view” event using your web analytics tool whenever a new page loads.

The URLSearchParams interface defines utility methods to work with the query string of a URL.

The URLSearchParams API provides a consistent interface to the bits and pieces of the URL and allows trivial manipulation of the query string (that stuff after "?").

```
const location = useLocation();
const queryParams = new URLSearchParams(location.search);
const isSortAsc = queryParams.get("sort") === "asc";
```

For eg url:http://localhost:3000/quotes?sort=asc

useRouteMatch()

The useRouteMatch hook attempts to match the current URL in the same way that a <Route> would. It's mostly useful for getting access to the match data without actually rendering a <Route>.

The useRouteMatch hook either:

- takes no argument and returns the match object of the current <Route>
- takes a single argument, which is identical to props argument of matchPath. It can be either a pathname as a string (like the example above) or an object with the matching props that Route accepts.

```
const match = useRouteMatch({
  path: "/BLOG/:slug/",
  strict: true,
  sensitive: true
});
```

- Provides access to the match object
- If it is provided with no arguments, it returns the closest match in the component or its parents.
- A primary use case would be to construct nested paths.

```
import { useRouteMatch, Route } from 'react-router-dom';
function Auth() {
  const match = useRouteMatch();
  return (
    <>
      <Route path={`${match.url}/login`}>
        <Login />
      </Route>
      <Route path={`${match.url}/register`}>
        <Register />
      </Route>
    </>
  );
}
```

8.1.6 <Redirect/>

In some cases, when the user clicks on a particular link, i.e the URL is pointing at a particular route, we want the user to be redirected to a different page or route.

We will see the following props associated with Redirect:

to: string :- The URL to redirect to. Any valid URL path that path-to-regexp@^1.7.0 understands. All URL parameters that are used in to must be covered by from.

We use **to** for redirecting from a particular route path to a different path.

8.1.7 <Switch>

```
<Route exact path='/' component={Home} />

<Route path='/about' component={About} />

<Route path='/contact' component={ContactUs} />

<Route path='/:notMatching' component={NotFound} />
```

In the above example, NotFound component will be rendered for all the routes except for the Home component as 'exact' path is mentioned for it.

If the URL is localhost:3000/login or localhost:3000/about, it will be matched with /:notMatching path as well, hence NotFound component will also be rendered.

But this is not the behavior we are expecting. We need only one Route to be rendered based on the first matched Route rather than matching all the Route paths. For this requirement, we can make use of the **<Switch />** component provided by **react-router-dom** library.

The **<Switch />** component returns only the first matching Route rather than checking all matching Routes. The <Switch/> component, will wrap all the Routes and will pick only the first matching Route among all the configured Routes. Once the Route is matched, it stops looking for the other Routes and renders only the matched Route component.

```
<Switch>

 <Route exact path='/' component={Home} />

 <Route path='/about' component={About} />

 <Route path='/contact' component={ContactUs} />

 <Route path='/:notMatching' component={NotFound} />

</Switch>
```

8.1.8 <Prompt>

Used to prompt the user before navigating away from a page. When your application enters a state that should prevent the user from navigating away (like a form is half-filled out), render a <Prompt>.

- message: string

The message to prompt the user with when they try to navigate away.

- message: func

Will be called with the next location and action the user is attempting to navigate to. Return a string to show a prompt to the user or true to allow the transition.

- when: bool

Instead of conditionally rendering a <Prompt> behind a guard, you can always render it but pass when={true} or when={false} to prevent or allow navigation accordingly.

```
<Prompt when={formIsHalfFilledOut} message="Are you sure?" />
```

```
<Prompt
  message={(location, action) => {
    if (action === 'POP') {
      console.log("Backing up...")
    }

    return location.pathname.startsWith("/app")
      ? true
      : `Are you sure you want to go to ${location.pathname}?`
  }}
/>
```

9. Lazy Loading

Most React apps will have their files “bundled” using tools like Webpack, Rollup or Browserify.

Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

If you’re using Create React App, Next.js, Gatsby, or a similar tool, you will have a Webpack setup out of the box to bundle your app.

9.1 Code Splitting

Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don’t accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it’s good to get ahead of the problem and start “splitting” your bundle. Code-Splitting is a feature supported by bundlers like Webpack, Rollup and Browserify (via factor-bundle) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you “lazy-load” just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven’t reduced the overall amount of code in your app, you’ve avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

9.1.1 import()

The best way to introduce code-splitting into your app is through the dynamic import() syntax.

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you’re using Create React App, this is already configured for you and you can start using it immediately. It’s also supported out of the box in Next.js.

If you’re setting up Webpack yourself, you’ll probably want to read Webpack’s guide on code splitting. Your Webpack config should look vaguely like this.

When using Babel, you’ll need to make sure that Babel can parse the dynamic import syntax but is not transforming it. For that you will need @babel/plugin-syntax-dynamic-import.

9.1.2 React.lazy

The React.lazy function lets you render a dynamic import as a regular component.

Before:

```
import OtherComponent from './OtherComponent';
```

After:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

This will automatically load the bundle containing the OtherComponent when this component is first rendered.

React.lazy takes a function that must call a dynamic import(). This must return a Promise which resolves to a module with a default export containing a React component.

The 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.

The fallback prop accepts any React elements that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap multiple lazy components with a single Suspense component.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

10. Animating React Apps

React is a library for building user interfaces. For a React frontend developer, implementing animations on webpages is an integral part of your daily work, from animating text or images to complex 3D animations. Animations can help to improve the overall user experience while building a React application.

Ways to animate a ReactJS app:

10.1 CSS Method

```
import React from 'react';

import './Modal.css';

const modal = (props) => {
   const cssClasses=["Modal",props.show? "ModalOpen":"ModalClose"]
   return (<div className={cssClasses.join(" ")}>
      <h1>A Modal</h1>
      <button className="Button" onClick={props.closed}>Dismiss</button>
   </div>)
};

export default modal;
```

```
.Modal {
   position: fixed;
   z-index: 200;
   border: 1px solid #eee;
   box-shadow: 0 2px 2px #ccc;
   background-color: white;
   padding: 10px;
   text-align: center;
   box-sizing: border-box;
   top: 30%;
   left: 25%;
   width: 50%;
   transition: all 300ms ease-out;
}

.ModalOpen{

animation:openModal 400ms ease-out forwards;
}

.ModalClose{

   animation: closeModal 400ms ease-out forwards;
}

@keyframes openModal {
   0%{
        opacity: 0;
        transform: translateY(-100%);

    }
    50%{
        opacity: 0;
        transform: translateY(90%);

    }
    100%{
        opacity: 1;
        transform: translateY(0);
    }
}

@keyframes closeModal {
    0%{
        opacity: 1;
        transform: translateY(0);
    }
    50%{
        opacity: 0.8;
        transform: translateY(60%);

    }
    100%{
        opacity: 0;
        transform: translateY(-100%);
    }
}
```

```
import React, { Component } from "react";

import "./App.css";
import Modal from "./components/Modal/Modal";
import Backdrop from "./components/Backdrop/Backdrop";
import List from "./components/List/List";

class App extends Component {
  state={
    modalIsOpen:false
  }

  showModal=()=>{
    this.setState({modalIsOpen:true})
  }

  closeModal=()=>{
    this.setState({modalIsOpen:false})
  }

  render() {
    return (
      <div className="App">
        <h1>React Animations</h1>
        <Modal show={this.state.modalIsOpen} closed={this.closeModal}/>
        <Backdrop show={this.state.modalIsOpen} />
        <button className="Button" onClick={this.showModal}>Open Modal</button>
        <h3>Animating Lists</h3>
        <List />
      </div>
    );
  }
}
export default App;
```

10.2 React Transition Group

It uses as a low-level API for animation.

While React manages different components and their behavior to make any web app interactive it contains no methodology of adding aesthetics to the app itself. The animation is considered to be one of the most used methods to add aesthetics to an app, we can add animation to a React App using an explicit group of components known as the **React Transition Group**.

The developers of the Transition Group define it as,

A set of components for managing component states (including mounting and unmounting) over time, specifically designed with animation in mind.

Anyway, things you need to know about it add-on component:

- React Transition Group changes the classes when component lifecycle changes. In turn, animated style should be described in CSS classes.
- The animation that we can implement using the React Transition Group is pure CSS Transitions, i.e. it doesn't use any property of JavaScript to animate the components. The animations are accomplished by defining classes with varying CSS styles and equipping and unequipping the classes at specified points in the lifecycle of the component.
- React was meant to be used to create web apps and the developers thought it would be best to separate the other additional features such as animations to keep react light weighted. Thus in order to use the Transition Group, we will need to install it separately.
- React Transition group consists of three primary components Transition, CSSTransition and TransitionGroup
- React Transition Group consists only of components i.e. in order to implement animation to any set of components or HTML elements we must firstly wrap them within any of the three existing components.

You can install transition group in your project component by using the command:

```
npm install react-transition-group --save
```

You can import transition component in project component by using the command:

```
import { Transition } from 'react-transition-group'
```

10.2.1 Components of React Transition Group:

10.2.1.1 Transition

The Transition component lets you describe a transition from one component state to another *over time* with a simple declarative API. Most commonly it's used to animate the mounting and unmounting of a component, but can also be used to describe in-place transition states as well.

You can import transition component in project component by using the command:

```
    import { Transition } from 'react-transition-group';
```

Note: Transition is a platform-agnostic base component. If you're using transitions in CSS, you'll probably want to use CSSTransition instead. It inherits all the features of Transition, but contains additional features necessary to play nice with CSS transitions (hence the name of the component).

By default the Transition component does not alter the behavior of the component it renders, it only tracks "enter" and "exit" states for the components. It's up to you to give meaning and effect to those states.

There are 4 main states a Transition can be in:

- 'entering'
- 'entered'
- 'exiting'
- 'exited'

Transition state is toggled via the in prop. When true the component begins the "Enter" stage. During this stage, the component will shift from its current transition state, to 'entering' for the duration of the transition and then to the 'entered' stage once it's complete.

When in is false the same thing happens except the state moves from 'exiting' to 'exited'.

```
<Transition in={inProp} timeout={500}>

      {state => (

        // ...

      )}

    </Transition>
```

children

A function child can be used instead of a React element. This function is called with the current transition status ('entering', 'entered', 'exiting', 'exited'), which can be used to apply context specific props to a component.

type: Function | element

required

in

Show the component; triggers the enter or exit states

type: boolean

default: false

mountOnEnter

By default the child component is mounted immediately along with the parent Transition component. If you want to "lazy mount" the component on the first in={true} you can set mountOnEnter. After the first enter transition the component will stay mounted, even on "exited", unless you also specify unmountOnExit.

type: boolean

default: false

unmountOnExit

By default the child component stays mounted after it reaches the 'exited' state. Set unmountOnExit if you'd prefer to unmount the component after it finishes exiting.

type: boolean

default: false

type: boolean

default: true

timeout

The duration of the transition, in milliseconds. Required unless addEndListener is provided.

You may specify a single timeout for all transitions:

timeout={500}

or individually:

```
timeout={{
 appear: 500,
 enter: 300,
 exit: 500,
}}
```

- appear defaults to the value of enter
- enter defaults to 0
- exit defaults to 0

```
import React from "react";
import Transition from "react-transition-group/Transition";

import "./Modal.css";

const animationTiming = {
    enter: 400,
    exit: 1000
};

const modal = props => {
  return (
    <Transition
        mountOnEnter
        unmountOnExit
        in={props.show}
        timeout={animationTiming}>
      {state => {
        const cssClasses = [
          "Modal",
          state === "entering"
            ? "ModalOpen"
            : state === "exiting" ? "ModalClose" : null
        ];
        return (
          <div className={cssClasses.join(" ")}>
            <h1>A Modal</h1>
            <button className="Button" onClick={props.closed}>
              Dismiss
            </button>
          </div>
        );
      }}
    </Transition>
  );
};

export default modal;
```

```
.Modal {
   position: fixed;
   z-index: 200;
   border: 1px solid #eee;
   box-shadow: 0 2px 2px #ccc;
   background-color: white;
   padding: 10px;
   text-align: center;
   box-sizing: border-box;
   top: 30%;
   left: 25%;
   width: 50%;
   transition: all 0.3s ease-out;
}

.ModalOpen {
   animation: openModal 0.4s ease-out forwards;
}

.ModalClose {
   animation: closeModal 0.4s ease-out forwards;
}

@keyframes openModal {
   0% {
      opacity: 0;
      transform: translateY(-100%);
   }
   50% {
      opacity: 1;
      transform: translateY(90%);
   }
   100% {
      opacity: 1;
      transform: translateY(0);
   }
}

@keyframes closeModal {
   0% {
      opacity: 1;
      transform: translateY(0);
   }
   50% {
      opacity: 0.8;
      transform: translateY(60%);
   }
   100% {
      opacity: 0;
      transform: translateY(-100%);
   }}
```

10.2.1.2 CSSTransition

A transition component inspired by the excellent ng-animate library, you should use it if you're using CSS transitions or animations. It's built upon the Transition component, so it inherits all of its props.

CSSTransition applies a pair of class names during the appear, enter, and exit states of the transition. The first class is applied and then a second *-active class in order to activate the CSS transition. After the transition, matching *-done class names are applied to persist the transition state.

```
function App() {
  const [inProp, setInProp] = useState(false);
  return (
    <div>
      <CSSTransition in={inProp} timeout={200} classNames="my-node">
        <div>
          {"I'll receive my-node-* classes"}
        </div>
      </CSSTransition>
      <button type="button" onClick={() => setInProp(true)}>
        Click to Enter
      </button>
    </div>
  );
}
```

When the in prop is set to true, the child component will first receive the class example-enter, then the example-enter-active will be added in the next tick. CSSTransition forces a reflow between before adding the example-enter-active. This is an important trick because it allows us to transition between example-enter and example-enter-active even though they were added immediately one after another. Most notably, this is what makes it possible for us to animate appearance.

classNames

The animation classNames applied to the component as it appears, enters, exits or has finished the transition. A single name can be provided, which will be suffixed for each stage, e.g. classNames="fade" applies:

- fade-appear, fade-appear-active, fade-appear-done
- fade-enter, fade-enter-active, fade-enter-done
- fade-exit, fade-exit-active, fade-exit-done

A few details to note about how these classes are applied:

1. They are *joined* with the ones that are already defined on the child component, so if you want to add some base styles, you can use className without worrying that it will be overridden.
2. If the transition component mounts with in={false}, no classes are applied yet. You might be expecting *-exit-done, but if you think about it, a component cannot finish exiting if it hasn't entered yet.
3. fade-appear-done and fade-enter-done will *both* be applied. This allows you to define different behavior for when appearing is done and when regular entering is done, using selectors like .fade-enter-done:not(.fade-appear-done). For example, you could apply an epic entrance animation when element first appears in the DOM using Animate.css. Otherwise you can simply use fade-enter-done for defining both cases.

Each individual classNames can also be specified independently like:

```
classNames={{
 appear: 'my-appear',
 appearActive: 'my-active-appear',
 appearDone: 'my-done-appear',
 enter: 'my-enter',
 enterActive: 'my-active-enter',
 enterDone: 'my-done-enter',
 exit: 'my-exit',
 exitActive: 'my-active-exit',
 exitDone: 'my-done-exit',
}}
```

If you want to set these classes using CSS Modules:

import styles from './styles.css';

you might want to use camelCase in your CSS file, that way could simply spread them instead of listing them one by one:

classNames={{ ...styles }}

type: string | { appear?: string, appearActive?: string, appearDone?: string, enter?: string, enterActive?: string, enterDone?: string, exit?: string, exitActive?: string, exitDone?: string, }

default: ''

10.2.1.3 TransitionGroup

The <TransitionGroup> component manages a set of transition components (<Transition> and <CSSTransition>) in a list. Like with the transition components, <TransitionGroup> is a state machine for managing the mounting and unmounting of components over time.

Note that <TransitionGroup> does not define any animation behavior! Exactly how a list item animates is up to the individual transition component. This means you can mix and match animations across different list items**.**

component

<TransitionGroup> renders a <div> by default. You can change this behavior by providing a component prop. If you use React v16+ and would like to avoid a wrapping <div> element you can pass in component={null}. This is useful if the wrapping div borks your css styles.

type: any

default: 'div'

```
import React, { Component } from "react";
import { CSSTransition, TransitionGroup } from "react-transition-group";
import "./List.css";

class List extends Component {
  state = {
    items: [1, 2, 3],
  };

  addItemHandler = () => {
    this.setState((prevState) => {
      return {
        items: prevState.items.concat(prevState.items.length + 1),
      };
    });
  };

  removeItemHandler = (selIndex) => {
    this.setState((prevState) => {
      return {
        items: prevState.items.filter((item, index) => index !== selIndex),
      };
    });
  };

  render() {
    const listItems = this.state.items.map((item, index) => (
      <CSSTransition key={index} classNames="fade" timeout={300}>
        <li className="ListItem" onClick={() => this.removeItemHandler(index)}>
          {item}
        </li>
      </CSSTransition>
    ));
    return (
      <div>
        <button className="Button" onClick={this.addItemHandler}>
          Add Item
        </button>
        <p>Click Item to Remove.</p>
        <TransitionGroup component="ul" className="List">
          {listItems}
        </TransitionGroup>
      </div>
    );
  }
}

export default List;
```

```
.List {
  list-style: none;
  margin: 0 auto;
  padding: 0;
  width: 280px;
}

.ListItem {
  margin: 0;
  padding: 10px;
  box-sizing: border-box;
  width: 100%;
  border: 1px solid #521751;
  background-color: white;
  text-align: center;
  cursor: pointer;
}

.ListItem:hover,
.ListItem:active {
  background-color: #ccc;
}

.fade-enter {
  opacity: 0;
}

.fade-enter-active {
  opacity: 1;
  transition: opacity 0.3s ease-out;
}

.fade-exit {
  opacity: 1;
}
```

```
.fade-exit-active {
  opacity: 0;
  transition: opacity 0.3s ease-out;
}
```

11. Testing

11.1 Need for testing component

Consider a counter application. In a counter, we have a counter state and two functions, increment and decrement. Increment function will increase counter by one and decrement function will decrease counter by one.

In order to know that increment and decrement is working properly in counter app, we have to test them. To make sure that functions and component are working properly, we need testing.

Writing tests is a key aspect of any software development. Writing test cases for component in React helps in many aspect for building better React application.

Few are as follows:

- Helps in verifying automatically that component is working properly
- Test cases ensure the quality of React component
- It helps in identifying unexpected changes in component during development
- It helps in detecting bugs in early stage of development which reduce cost, developer effort and well as makes process more agile.

It is necessary that we test working of our React component. We might not need to test every component, but we should test all critical components of an application.

11.2 React Testing Library

The DOM Testing Library is a very light-weight solution for testing DOM nodes (whether simulated with JSDOM as provided by default with Jest or in the browser). The main utilities it provides involve querying the DOM for nodes in a way that's similar to how the user finds elements on the page. In this way, the library helps ensure your tests give you confidence in your UI code.

React Testing Library builds on top of DOM Testing Library by adding APIs for working with React components.

The React Testing Library is a very light-weight solution for testing React components. It provides light utility functions on top of react-dom and react-dom/test-utils, in a way that encourages better testing practices.

11.2.1 Render

const {/* */} = render(Component) returns:

- unmount function to unmount the component
- container reference to the DOM node where the component is mounted
- all the queries from DOM Testing Library, bound to the document so there is no need to pass a node as the first argument (usually, you can use the screen import instead)

11.2.2 Queries

Difference from DOM Testing Library

The queries returned from render in React Testing Library are the same as DOM Testing Library except they have the first argument bound to the document, so instead of getByText(node, 'text') you do getByText('text')

Types of Queries

11.2.2.1 Single Elements

- getBy...: Returns the matching node for a query, and throw a descriptive error if no elements match or if more than one match is found (use getAllBy instead if more than one element is expected).
- queryBy...: Returns the matching node for a query, and return null if no elements match. This is useful for asserting an element that is not present. Throws an error if more than one match is found (use queryAllBy instead if this is OK).
- findBy...: Returns a Promise which resolves when an element is found which matches the given query. The promise is rejected if no element is found or if more than one element is found after a default timeout of 1000ms. If you need to find more than one element, use findAllBy.

11.2.2.2 Multiple Elements

- getAllBy...: Returns an array of all matching nodes for a query, and throws an error if no elements match.
- queryAllBy...: Returns an array of all matching nodes for a query, and return an empty array ([]) if no elements match.
- findAllBy...: Returns a promise which resolves to an array of elements when any elements are found which match the given query. The promise is rejected if no elements are found after a default timeout of 1000ms.
- findBy methods are a combination of getBy* queries and waitFor. They accept the waitFor options as the last argument (i.e. await screen.findByText('text', queryOptions, waitForOptions))

11.2.2.3 Priority

Based on the Guiding Principles, your test should resemble how users interact with your code (component, page, etc.) as much as possible. With this in mind, we recommend this order of priority:

11.2.2.3.1 Queries Accessible to Everyone

Queries that reflect the experience of visual/mouse users as well as those that use assistive technology.

- getByRole: This can be used to query every element that is exposed in the accessibility tree. With the name option you can filter the returned elements by their accessible name. This should be your top preference for just about everything. There's not much you can't get with this (if you can't, it's possible your UI is inaccessible). Most often, this will be used with the name option like so: getByRole('button', {name: /submit/i}). Check the list of roles.
- getByLabelText: This method is really good for form fields. When navigating through a website form, users find elements using label text. This method emulates that behavior, so it should be your top preference.
- getByPlaceholderText: A placeholder is not a substitute for a label. But if that's all you have, then it's better than alternatives.
- getByText: Outside of forms, text content is the main way users find elements. This method can be used to find non-interactive elements (like divs, spans, and paragraphs).
- getByDisplayValue: The current value of a form element can be useful when navigating a page with filled-in values.

11.2.2.3.2 Semantic Queries

HTML5 and ARIA compliant selectors. Note that the user experience of interacting with these attributes varies greatly across browsers and assistive technology.

- getByAltText: If your element is one which supports alt text (img, area, and input), then you can use this to find that element.
- getByTitle: The title attribute is not consistently read by screenreaders, and is not visible by default for sighted users
- getByTestId: The user cannot see (or hear) these, so this is only recommended for cases where you can't match by role or text or it doesn't make sense (e.g. the text is dynamic).
- User-Event: user-event is a companion library for Testing Library that provides more advanced simulation of browser interactions than the built-in fireEvent method.

11.3 Tools for Testing

11.3.1 Jest

Jest is an open source, zero configuration testing tool created by Facebook, which is used to test Javascript code including React components. Facebook itself uses jest for testing React applications. Jest also provide snapshot testing which make it more suitable for React application.

Test file naming convention for Jest:

Since jest is zero configuration testing tool, we do not need to specify any configuration to locate test files. Any files which is placed inside **src/__tests__** directory or file name having extension **.spec.js** or **.test.js** will be automatically captured as test file by jest.

11.3.1.1 Expect

When you're writing tests, you often need to check that values meet certain conditions. expect gives you access to a number of "matchers" that let you validate different things.

Few commonly used Jest matchers are mentioned below table:

Method	Description
toBe (expected)	expect the actual value to be === to the expected value.
toBeDefined ()	expect the actual value to be defined. (Not undefined)
toBeFalsy ()	expect the actual value to be false
toBeTruthy ()	expect the actual value to be true.
toBeGreaterThan (expected)	expect the actual value to be greater than the expected value.
toBeLessThan (expected)	expect the actual value to be less than the expected value.
toEqual (expected)	expect the actual value to be equal to the expected, using deep equality comparison
toMatch (expected)	expect the actual value to match a regular expression
toThrow (expected)	expect a function to throw something.

You can find complete list of Jest matchers here.

11.3.1.2 Mock Functions

Mock functions are also known as "spies", because they let you spy on the behavior of a function that is called indirectly by some other code, rather than only testing the output. You can create a mock function with jest.fn(). If no implementation is given, the mock function will return undefined when invoked.

mockFn.mockResolvedValue(value)

Syntactic sugar function for:

jest.fn().mockImplementation(() => Promise.resolve(value));

Useful to mock async functions in async tests:

```
test('async test', async () => {
  const asyncMock = jest.fn().mockResolvedValue(43);

  await asyncMock(); // 43
});
```

Demo:

Greeting.js

```
import { useState } from "react"
import Output from "./Output"

const Greeting=()=>{
    const [text,setText]=useState(false)
    const changeTextHandler=()=>{
        setText(true)
    }
    return <div>
        <h2>Hello World</h2>
        {!text && <Output>Its good to see you</Output>}
        {text && <Output>Changed</Output>}
        <button onClick={changeTextHandler}>Change Text</button>
    </div>

}

export default Greeting
```

Greeting.test.js

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import Greeting from "./Greeting";
describe("<Greeting> component",()=>{
test('render hello world', () => {
  render(<Greeting />);
  const linkElement = screen.getByText(/Hello World/);
  expect(linkElement).toBeInTheDocument();
})
test('render Its good to see you', () => {
    render(<Greeting />);
    const linkElement = screen.getByText(/Its good to see you/);
    expect(linkElement).toBeInTheDocument();
  })
test('render Changed', () => {
    render(<Greeting />);
    userEvent.click(screen.getByRole("button"))
    const linkElement = screen.getByText(/Changed/);
    expect(linkElement).toBeInTheDocument();
  })
  test('does not render Its good to see you', () => {
    render(<Greeting />);
    userEvent.click(screen.getByRole("button"))
    const linkElement = screen.queryByText(/Its good to see you/);
    expect(linkElement).toBeNull();
  })
});
```

Async.js

```
import { useEffect, useState } from 'react';
const Async = () => {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then((response) => response.json())
      .then((data) => {
        setPosts(data);
      });
  }, []);

  return (
    <div>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
};
export default Async;
```

Async.test.js

```
import { render, screen } from '@testing-library/react';
import Async from './Async';
describe("Async Component",()=>{
test("render posts if request succeeds",async ()=>{
   window.fetch=jest.fn();
   window.fetch.mockResolvedValueOnce({
      json:async()=>[{id:"p1",title:"First Post"}]
   })
   render(<Async/>)
   const listitemElements=await screen.findAllByRole("listitem")
   expect(listitemElements).not.toHaveLength(0);
}

)

}
)
```