What are single page applications?

Single page applications are applications that donot reload whenever there is some changes in the page. Only the changed component reloads and not the whole website

Advantages of react

1. Single page application
2. Component based archetecture
3. Virtual DOM

React V/S Angular

React	Angular
React and Angular both are used to create single page UI applications using components.	
1. React is a JavaScript library	Angular is a complete framework.
2. React uses a **virtual DOM** which makes it faster.	Angular uses a real DOM
3. React is smaller in size and **lightweight** and therefore faster sometime.	Angular is bigger because it is a complete framework.
4. React depends on external libraries for many complex features, so developer has to write many lines of code for complex functionalities.	Since Angular is a complete framework, therefore it provide **built-in support** for features like routing, forms, validation, and HTTP requests.
5. React is **simple to learn** and more popular than Angular.	5. Angular is slightly difficult to learn as it has Typescript, OOPS concept and many more thing.

Components : Isolated, Indipendent and Reusable piece of code that return a JSX Template.
It is the fundamental block of any react application

Components

Class based components

Before useState: Statelfull

Function based Components

Stateless

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      date: new Date()
    }
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.props.name}</h1>
        <h2>It is {this.state.date.
        toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Class component

Creating State

Creating Component

```
import { useState } from "react"

export default function Clock({ name }) {
  const [date, setDate] = useState(new Date())

  return (
    <div>
      <h1>Hello, {name}</h1>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  )
}
```

Function component

Accessing Props: this.props.name

Defining State:

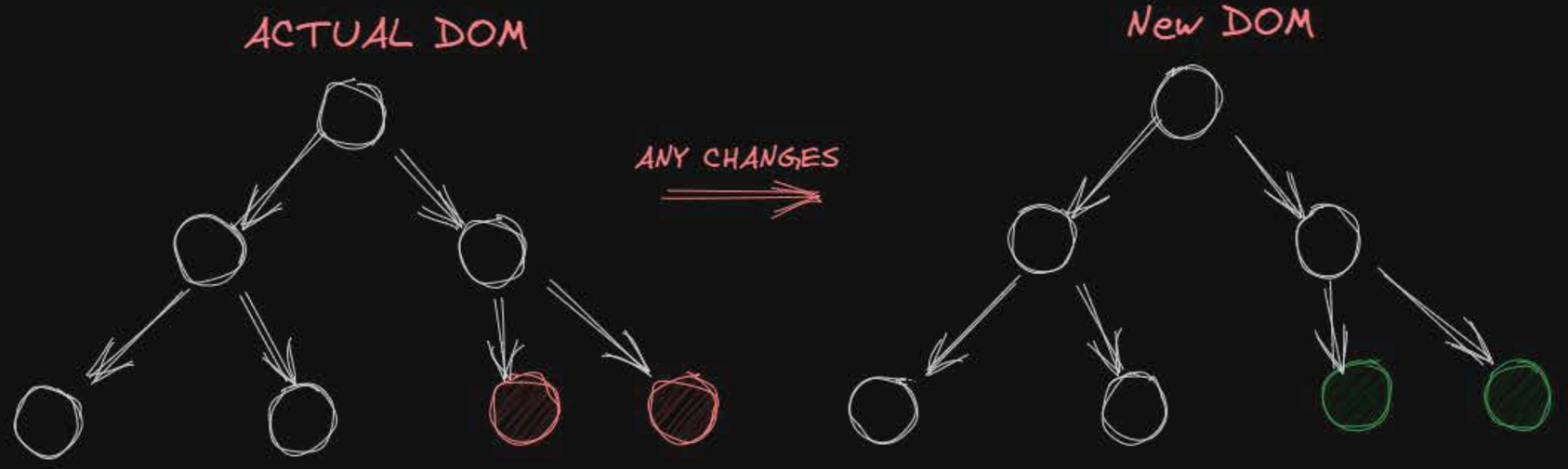
```
this.state{
    creating object
}
```

updating state:

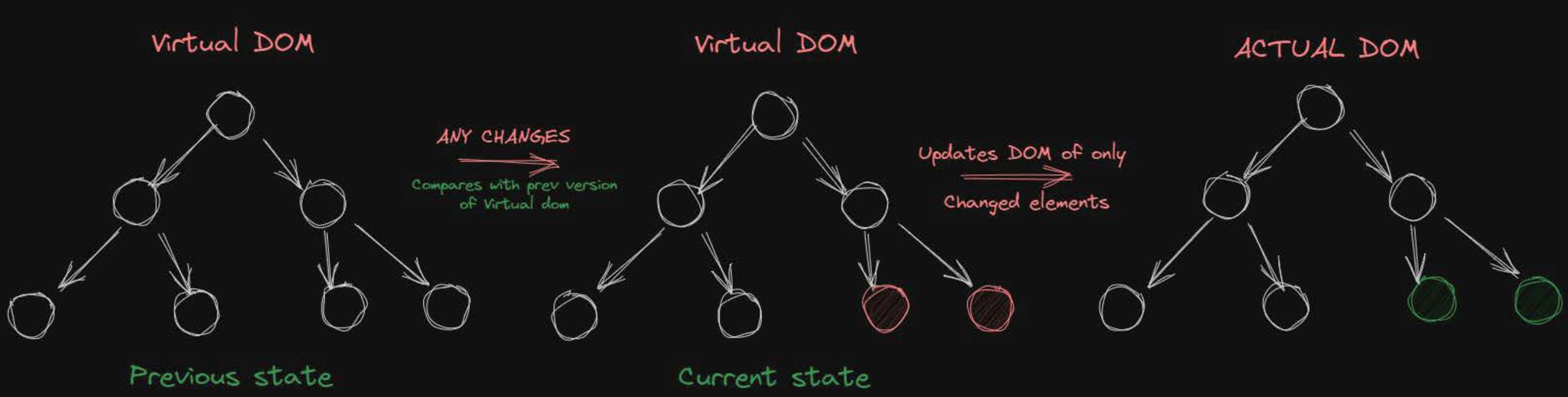
```
this.setState{
    newObject
}
```

```
props.name

const [state, setState] =
 useState(initial value);


setState(newVal);
```

Virtual Dom: Virtual representation of actual dom



Everytime there is some change in html dom, a completely new DOMis created in the browser which will repaint the UI again. UI Repaint is a very expensive process.



Whenever there is some change in the DOM, new virtual dom is created and it compares the current virtual dom with previous state of virtual dom and finds where changes occured.
I will update the actual dom only at those places

Conntrolled Input / Component

Unconntrolled Input / Component

Input validation

Conditionally disabling submit

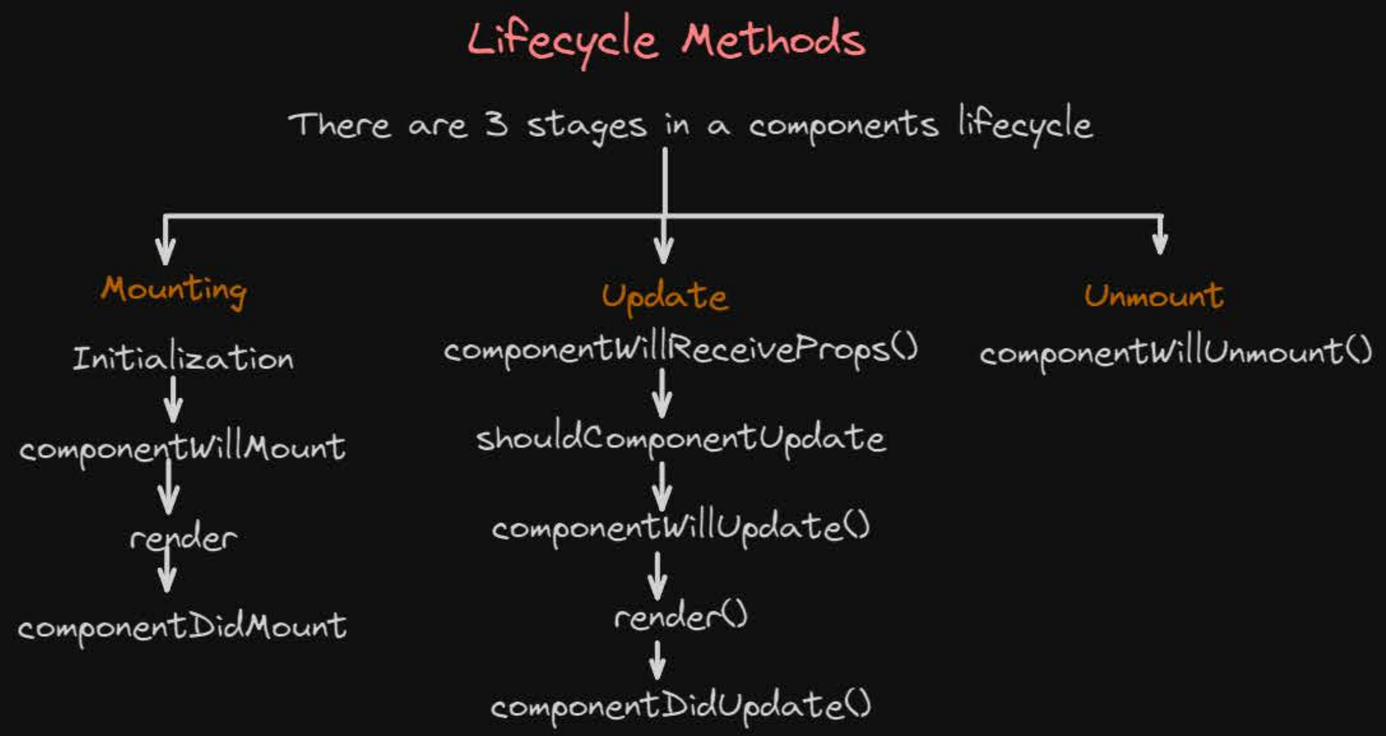
```
const [title, setTitle] = useState('')
const [date, setDate] = useState('')
```

```
<label>
  <span>Event Title:</span>
  <input
    type="text"
    onChange={(e) => setTitle(e.target.value)}
    value={title}
  />
</label>
```

```
const title = useRef()
const date = useRef()
```

```
const resetForm = () => {
  title.current.value = ""
  date.current.value = ""
}
```

```
<label>
  <span>Event Title:</span>
  <input
    type="text"
    ref={title}
  />
</label>
```



In React, lifecycle methods are methods that are called at different stages of a component's lifecycle. These methods allow you to perform specific actions or side effects during different phases of a component's existence.

Here is a list of the main lifecycle methods in class-based components:

Mounting Phase:
constructor(): This is the first method called when the component is created. It is used to initialize the component's state and bind event handlers.
componentWillMount(): Called just before the component is rendered for the first time.
render(): The mandatory method that returns the JSX that will be rendered.
componentDidMount(): Called after the component is mounted (added to the DOM). It is commonly used for making AJAX requests or setting up subscriptions.\

Updating Phase:
componentWillReceiveProps(): Called when the component receives new props from its parent. It is deprecated and replaced by componentDidUpdate.
shouldComponentUpdate(): Called before rendering when new props or state are received. Used for performance optimization, to decide whether the component should update or not.
componentWillUpdate() (deprecated): Called just before the component is updated. Avoid using this method and prefer componentDidUpdate for side effects.
render(): The component is re-rendered if shouldComponentUpdate returned true.
componentDidUpdate(): Called after the component has updated (re-rendered). Useful for performing side effects after a component update.

Unmounting Phase:
componentWillUnmount(): Called just before the component is removed from the DOM. It should be used to clean up resources like subscriptions or timers.

Higher order Components: A Higher Order Component is a function that takes a component and returns a new component with added or enhanced functionality.

```jsx
import React, { useState } from "react";

const withLogin = (WrappedComponent) => {
  return (props) => {
    const [isLoggedIn, setIsLoggedIn] = useState(false);

    const toggleLogin = () => {
      setIsLoggedIn(!isLoggedIn);
    };

    return (
      <WrappedComponent
        isLoggedIn={isLoggedIn}
        toggleLogin={toggleLogin}
        {...props}
      />
    );
  };
};

const LoginButton = (props) => {
  return (
    <button onClick={props.toggleLogin}>
      {props.isLoggedIn ? "Logout" : "Login"}
    </button>
  );
};

const LoginButtonWithHOC = withLogin(LoginButton);
```

Here's a simple example of a HOC that adds a simple login feature to a component:

In this example, the withLogin HOC takes the LoginButton component as an argument and returns a new component with the added isLoggedIn and toggleLogin state and methods. The original LoginButton component is wrapped by the HOC and receives the added props

Limitation of class based component

```jsx
class App extends React.Component{
  constructor(){
    super();
    this.state={
      count:10
    }
  }
  render(){
    console.log("render");
  return (
    <div className="App">
      {this.state.count}
      <button onClick={()=>this.setState({count:10})}>Update State</button>
    </div>
  );
  }
}

export default App;
```

Problem here is that the component is re-rendering on every click of button even if the value of count didnt change

```jsx
class App extends React.PureComponent{
  constructor(){
    super();
    this.state={
      count:10
    }
  }
  render(){
    console.log("render");
  return (
    <div className="App">
      {this.state.count}
      <button onClick={()=>this.setState({count:10})}>Update State</button>
    </div>
  );
  }
}

export default App;
```

To solve this we just update
React.Component with React.PureComponent

To skip re-rendering a class component for same props and state, extend PureComponent instead of Component:

How does it work ?

This makes use of the shouldComponentUpdate() lifecycle method. This method will retuen false if the value of state is same as befiore

Limitation:

PureComponent only does shallow comparition ie if the state contains complex data structures, it wont prevent rerenders.
eg : instead of this.count = 10; if we create an object this.count = {val = 10}; and update it when the button is clicked with same value of val ie 10, it will still trigger a rerender.

Creating a React App

* Creating new react app

```
>npx create-react-app my-first-react-app
```

* To begin the development, run `npm start`

```
npm start
```

```
Local:            http://localhost:3000
On Your Network:  http://192.168.47.229:3000
```

* Installing React Router for Multi page Website

```
npm install react-router-dom
```

* Importing react router packages from react-router-dom in App.js

```
import { BrowserRouter, Route, Switch, NavLink, Redirect } from 'react-router-dom'
```

React Basics

What is React?

React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.

React is a tool for building UI components.

How does React Work?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

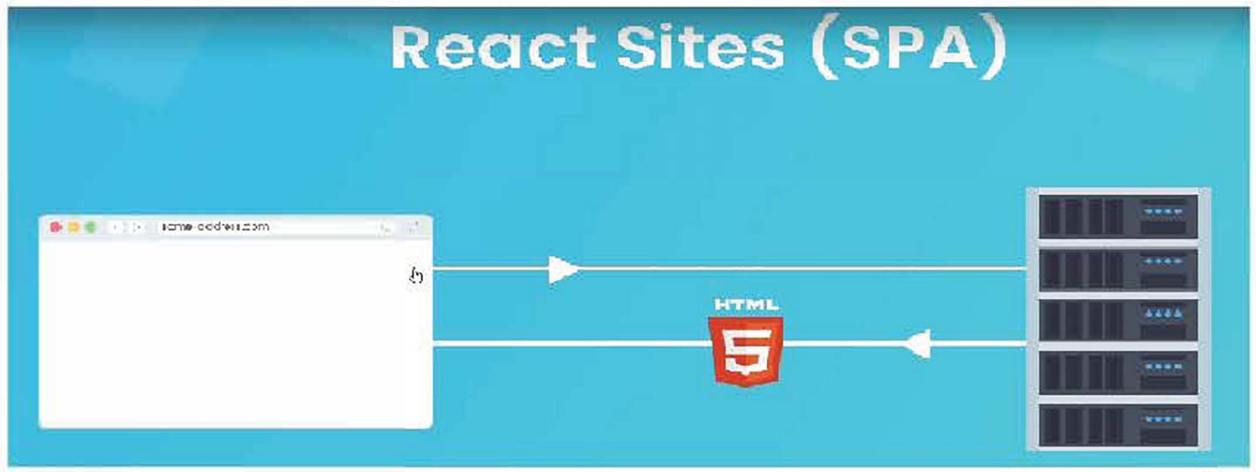
React only changes what needs to be changed!

React finds out what changes have been made, and changes **only** what needs to be changed.

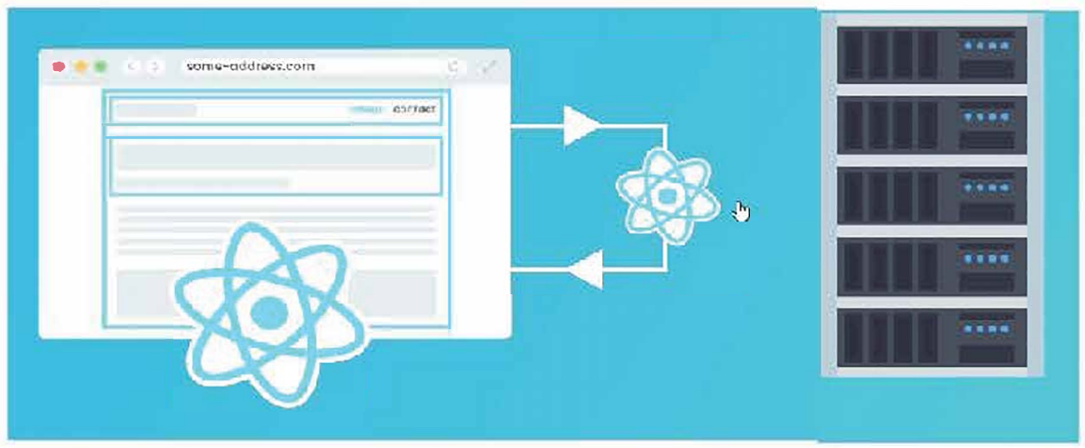
You will learn the various aspects of how React does this in the rest of this tutorial.

I a notmal HTML page, when you open a website, the browser would make a request to the server which returns the page

When you try to open another page on the website, the browser again sends a request to the server



The first step remains same in REACT WEBSITE as well



However when you try to navigate to another page on the same website REACT WOULD INTERCEPT THAT REQUEST THAT REQUEST TO THE SERVER PAGE.
THE REQUEST IS NOT SENT TO THE SERVER INSTEAD REACT LOOKS AT THE REQUEST AND THE PAGE WE WANT TO NAVIGATE TO AND IT LOADS THE CONTENT DYNAMICALLY INTO THE BROWSER

THIS RESULTS IN A MUCH SMOOTHER AND FASTER EXPERIENCE

BY SHIVKARAMAN

RENDER HTML

React renders HTML to the web page by using a function called ReactDOM.render().

The ReactDOM.render() function takes **two arguments**, HTML code and an HTML element.
The purpose of the function is to display the specified HTML code inside the specified HTML element.

But render where?
There is another folder in the root directory of your React project, named "public". In this folder, there is an index.html file.
You'll notice a single in the body of this file. This is where our React application will be rendered.

Example

Display a paragraph inside an element with the id of "root":

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

The result is displayed in the element:

```
<body>
  <div id="root"></div>
</body>
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>      You, 2 weeks ago • Initialize project us
);                                                          index.js
```

RENDER JSX

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.

JSX converts HTML tags into react elements.

ReactDOM.render(React Component , DOM element);

```
const Banner = () => {
  return (
    <div>
      <p>banner</p>
    </div>
  )
}

// render the component to the DOM
ReactDOM.render(<Banner />, document.getElementById('banner'))
```

React Component
name shoule start with capita letter
Returns a JSX template

A react component can contain two things
1. A JSX template
2. Component fucntionality in the form of JS funtions

```
<div id="banner">

</div>
```

```
▼<div>
    <p>banner</p> == $0
  </div>
```

BY SHIVKARAMAN

```
<script type="text/babel">
  const Banner = () => {
    return (
      <div>
        <p>banner</p>
      </div>
    )
  }

  // render the component to the DOM
  ReactDOM.render(<Banner />, document.getElementById('banner'))
</script>
```

Template which is present insiude the React component looks like HTML but is not actually HTML.
It is someting called as JSX (avascript XML)
It basically allows us to write HTML like syntax iside the JS file.

When this content is rendered to the DOM , that content is compiled into regular JS by the BABEL compiler and that JS injects the HTML (ie JSX) into the DOM.

This is the reason why we added BABEL earlier because browser doesnot understand JSX and it needs to be compiled to be runned.

One differnce betwen HTML and JSX is that in JSX we use className = "class-name" instead of class="class-name"

Second difference is that in JSX , it must have single root element

Template Expressions & Variables

```jsx
const Banner = () => {
  const title = "Welcome to my website"
  const subtitle = "this is a website for learning react"

  return (
    <div className="banner">
      <h2>{title.toUpperCase()}</h2>
      <p>{subtitle}</p>
    </div>
  )
}
```

We can also output expressions and variables in JSX template itself unlike the HTML

```jsx
const Banner = () => {
  const title = "Welcome to my website"
  const subtitle = "this is a website for learning react"
  const address = "https://www.netninja.dev"

  return (
    <div className="banner">
      <h2>{title.toUpperCase()}</h2>
      <p>{subtitle}</p>
      <p>Today's date is: {new Date().toDateString()}</p>
      <p>Random number: {Math.random() * 100}</p>
      <a href={address}>Visit the Net Ninja site</a>
    </div>
  )
}
```

We place dynamic values inside {}

Click And Event Handlers

```jsx
const Banner = () => {
  const title = "Welcome to my website"
  const subtitle = "this is a website for learning react"

  const handleClick = (e) => {
    console.log('you clicked me')
    console.log('the site title is:', title)
    console.log(e)
  }

  return (
    <div className="banner">
      <h2>{title.toUpperCase()}</h2>
      <p>{subtitle}</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  )
}
```

syntax

on'EventName' = {function}

Note: in onClick={handleEvent} , we are suppose to mention the function name only
If we use onClick={handleEvent( )} instead of the above syntax, the function haldleEvent will be called just after the react component gets evaluated

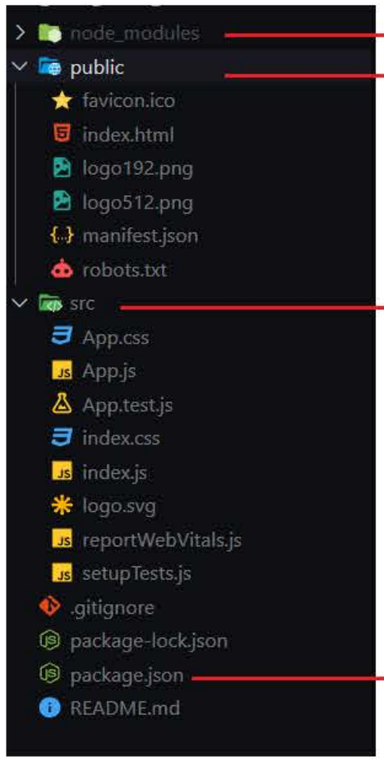
( ) -> means invoke here

So how do we pass arguments then? -> we use inline function
onClick={ ( ) => handleEvent (arg1, arg2, ..) }

```jsx
return (
  <div className="banner">
    <h2>{title.toUpperCase()}</h2>
    <p>{subtitle}</p>
    <button onClick={(e) => console.log(e)}>Click me</button>
  </div>
)
```

INLINE FUNCTION

USING create-react-app

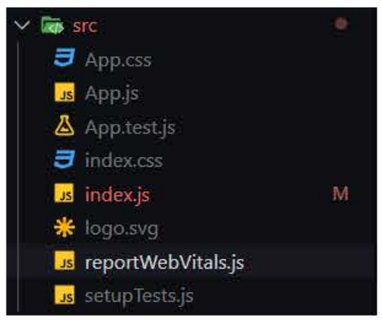


Stores all of the project dependencies and node packages. The react library itself is a node package and is kept in this folder. The additional packages that we will install in the future will also be stored in this folder.

The public folder contains files that is publid to the browser. The index.html is the sigle file that is served to the browser.
If we have a single page web app, this is the HTML present in there.
This contains a single root element where the react application will be injected.

This is where all the uncomplied react source code is gonna go.When our app is compiled, its all of these files that are gonna get compiled into JS bundle.

Keeps track of packages and dependencies ie the packages that we install



When you work with react, each component is gonna have its own file.
App.js is a file that contains code related to App react component.

App.css is a css file that is used to style the App.js component.

index.css is a global style sheet.

App.test.js is a file is for writing test for App.js ie App react component.

index.js is a file that is gonna kind of kickstart our react application.
Here we import React and ReactDom from 'react' and 'react-dom' packages and also react components from corresponding files

```
index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5
6  ReactDOM.render(
7    <React.StrictMode>
8      <App />
9    </React.StrictMode>,
10   document.getElementById('root')
11 );
```

StrictMode is an inbuilt react component that automatically checks for errors and displays it at runtime

```
>npx create-react-app my-first-react-app
```

state using normal variable

state : component data that may change over time

eg: In a todo list, if we add a new todo or delete a todo. In this case, the data inside the todos component is whats known as **component state** ie data inside the component that is changing

App.js > App > handleClick

```
1  import './App.css';
2
3  function App() {
4    let name = 'mario'
5
6    const handleClick = () => {
7      name = 'luigi'
8      console.log(name)
9    }
10
11   return (
12     <div className="App">
13       <h1>My name is {name}</h1>
14       <button onClick={handleClick}>Change name</button>
15     </div>
16   );
17 }
18
19 export default App;
```

When we click the buttin the fucntion is getting called but the content of the page is not getting altered to **My name is luigi** when the button is ckicked

This is because when our react application starts before the react component is even mounted to the DOM, react runs our component funtion -> COMPONENT IS BEING EVALUATED

The JSX content is compiled and injected into the HTML. When we press the button, it runs the function and updates the value of **name** but by this stage, the react component is already been fully evaluated and the template (which was present before) is already injected into the DOM. When we change the value of the variable, react doesnot re-evaluate the component.

For this to work, when we change the value of a variale, react need to re-evaluate the react component so that we can see that new value

But when we use regular variables like this and change them, that doesnt happen. Those variable changes dont trigger re-evaluation of react component. Hence we create states in a special way such that a state change innvokes revaluation of the react component

useState

We use useState to overcome the problem discussed earier.

useState is a react hook (**def :** Hooks allow us to "hook" into React features such as state and lifecycle methods. Hooks are functions that you can use inside functional components to add additional functionality.) all react hooks starts with a keywork use

useState hook is used to create a component state and whenever that state value changes, it triggers re-evaluation the react component

```
import { useState } from 'react'
```

useState accepts an initial state and returns two values in an array

1. The current state.
2. A function that is used for updating the state

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

We can also pass a function as an argument if the initial state has to be computed. And the value returned by the function will be used as the initial state.

```
const [sum, setsum] = useState(function generateRandomInteger(){5+7);})
```

We cannot just update the value of state just like normal variable, we use have to use the **setState(newState)** function

App.js > App > handleClick

```
import './App.css';

function App() {
  let name = 'mario'

  const handleClick = () => {
    name = 'luigi'
    console.log(name)
  }

  return (
    <div className="App">
      <h1>My name is {name}</h1>
      <button onClick={handleClick}>Change name</button>
    </div>
  );
}
```

App.js > App

```
import './App.css'
import { useState } from 'react'

function App() {
  const [name, setName] = useState('mario')

  const handleClick = () => {
    setName('luigi')
    console.log(name)
  }

  return (
    <div className="App">
      <h1>My name is {name}</h1>
      <button onClick={handleClick}>Change name</button>
    </div>
  );
}
```

How state changing & rerendering works

INITIAL TEMPLATE EVALUATION

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

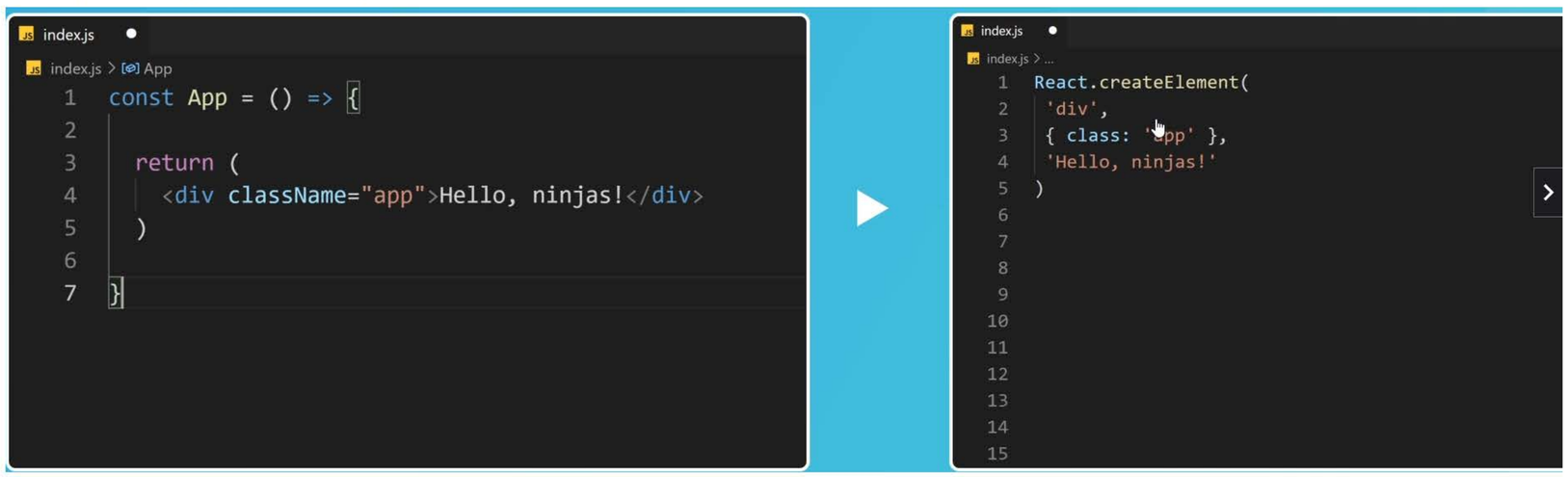
index.js is the entry point to the application and that runs first and it's this job to currently take our **app component** (according to our example) and render it to the DOM, but before it renders it to the DOM the 'app' component has to be evaluated and the JS template in that component has to be compiled .

So react evaluates our app components -> it runs the component function and looks at the JSX template that we return inside the components. It injects any dynamic values or state into the template and also hooks up any event listners like the handleClick() funtion .

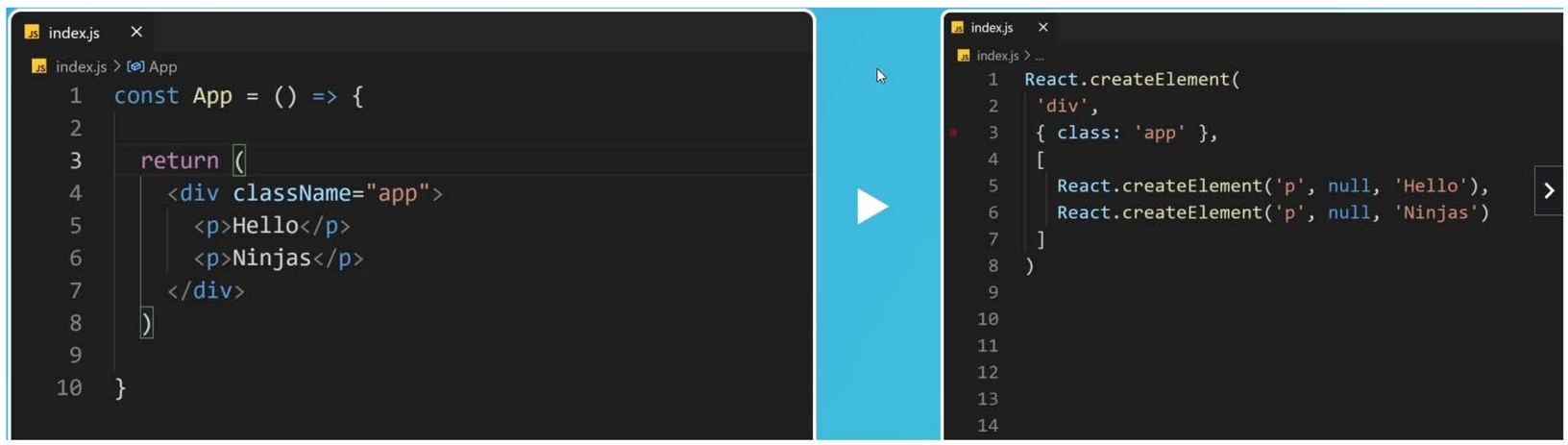
JSX needs to be compiled. JSX even though it's very very similar to HTML browsers don't know how to work with it . During the compile step well **JSX is compiled into JavaScript** it's not compiled into HTML straight away.

What does JSX look like when it's been compiled into JavaScript?

1. simple component template



2. complex component -> Uses nesting React.createElement



Once compiled, we will have a tree like representation of the JSX template and this gets passed into the render method of the ReactDom object whick looks into the tree like structure and uses JS DOM to inject the tree template into the webpage in the form of HTML

This tree like structure is called as the **VIRTUAL DOM**

RE-EVALUATION ON STATE CHANGE

When there is a state chnage, the new JSX template is recompiled into JS and then ReactDom looks into new JS treee template and compares it to the older one (of initial render).
The comparision is done to see what changes there are adn and only updates the real DOM only when it sees a change in the two virtual representations of the DOM.

THIS CYCLE CONTINUES WHENEVER THERE IS A STATE CHANGE

Outputting Lists

```
const [events, setEvents] = useState([
  {title: "mario's birthday bash", id: 1},
  {title: "bowser's live stream", id: 2},
  {title: "race on moo moo farm", id: 3}
])

return (
  <div className="App">
    {events.map((event) => (
      <div>
        <h2>{event.title}</h2>
      </div>
    ))}
  </div>
);
```

include key

```
function App() {
  const [events, setEvents] = useState([
    {title: "mario's birthday bash", id: 1},
    {title: "bowser's live stream", id: 2},
    {title: "race on moo moo farm", id: 3}
  ])

  return (
    <div className="App">
      {events.map((event, index) => (
        <div key={event.id}>
          <h2>{index} - {event.title}</h2>
        </div>
      ))}
    </div>
  );
}
```

mario's birthday bash

bowser's live stream

race on moo moo farm

```
▶ Warning: Each child in a list should have a unique "key" prop.    index.js:1

Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.
    at div
    at App (http://localhost:3000/main.f424b22….hot-update.js:31:81)
```

"key " property --> A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists. The next thing that comes to mind is that what should be good to be chosen as key for the items in lists. It is recommended to use a string as a key that uniquely identifies the items in the list.

using setEvents and adding event listner to list

```
function App() {
  const [events, setEvents] = useState([
    {title: "mario's birthday bash", id: 1},
    {title: "bowser's live stream", id: 2},
    {title: "race on moo moo farm", id: 3}
  ])

  const handleClick = (id) => {
    setEvents(events.filter((event) => {
      return id !== event.id
    }))
    console.log(id)
  }

  return (
    <div className="App">
      {events.map((event, index) => (
        <div key={event.id}>
          <h2>{index} - {event.title}</h2>
          <button onClick={() => handleClick(event.id)}>delete event</button>
        </div>
      ))}
    </div>
```

remove element on clicking button

0 - mario's birthday bash

delete event

1 - bowser's live stream

delete event

2 - race on moo moo farm

delete event

We have a small problem,
We have used **currState (events)** directly in the setEvent function which is not a good practice

Whenever there is a state change, the DOM is not updated right away, the update is scheduled by react for future.
Say the **events** state was updated elsewhere, by the time our update occurs the **events** state (the one that we have used in this setEvents func) may be outdated and this update might not work as expected

Insetead, we can pass a function in setEvents function

```
const handleClick = (id) => {
  setEvents((prevEvents) => {
    return prevEvents.filter((event) => {
      return id !== event.id
    })
  })
  console.log(id)
}
```

prevEvents -> refers to previous state

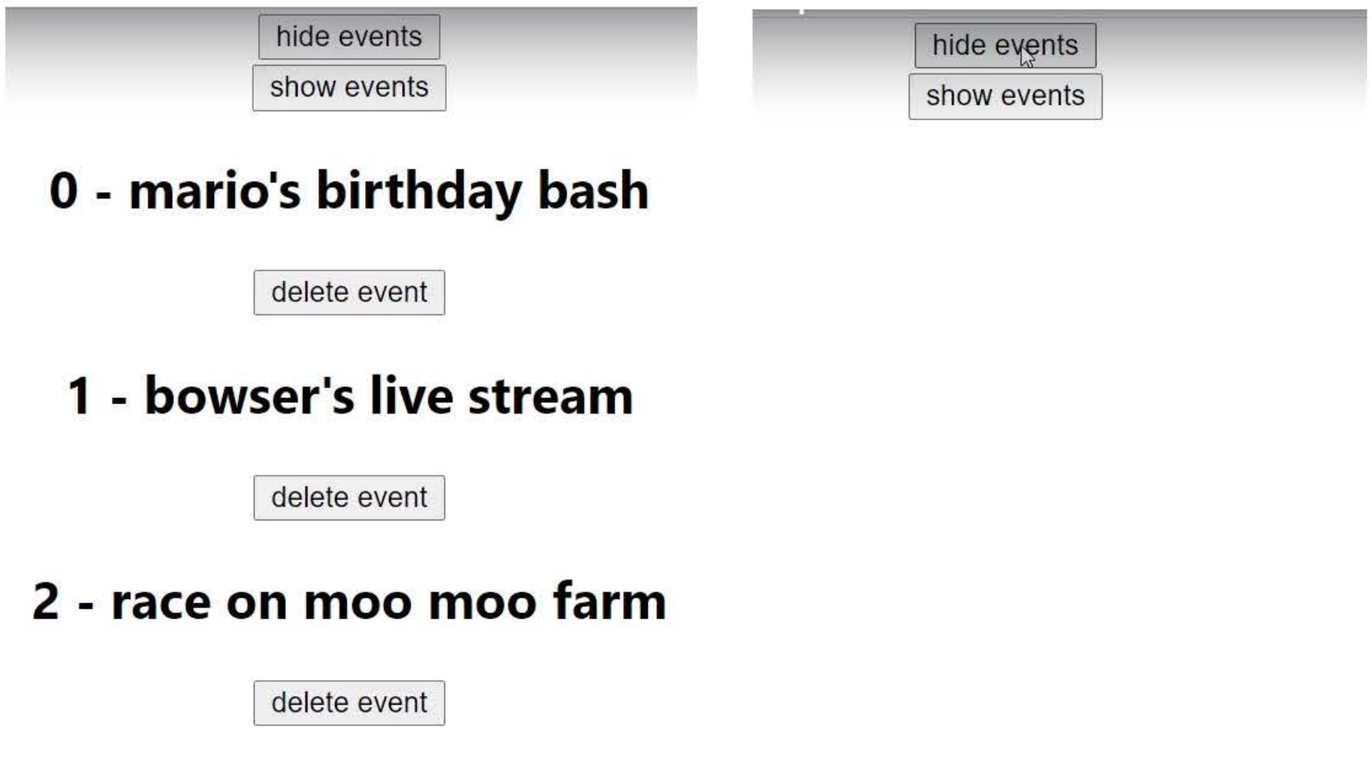
The function takes the value of previous state at the time of updation and that previous state valus is guarenteed to be up to date and we need not worry about that as in the above case

Conditinal template

In the previous example, we learnt how to render lists
Now anothe thing for which we can use state for in a component is to conditonally output part of a template based on that state
Eg: dropdown button -> if state is true, we show the content else hide the content

We update the prev example a little bit -> We set up 2 buttons **show events** and **hide events** and we use a new useState to suppoet this functionality

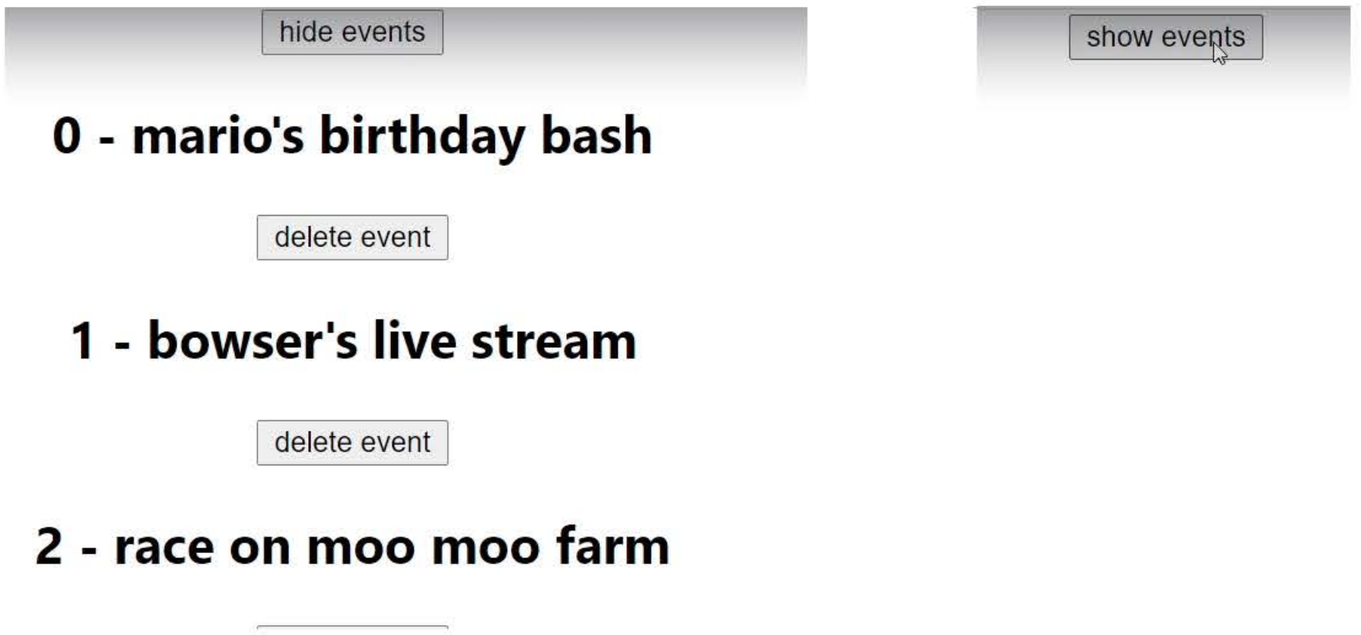
```jsx
function App() {
  const [showEvents, setShowEvents] = useState(true)
  const [events, setEvents] = useState([
    {title: "mario's birthday bash", id: 1},
    {title: "bowser's live stream", id: 2},
    {title: "race on moo moo farm", id: 3}
  ])
      You, 25 seconds ago • Uncommitted changes
  console.log(showEvents)

  const handleClick = (id) => {
    setEvents(prevEvents => {
      return prevEvents.filter(event => id !== event.id)
    })
  }

  return (
    <div className="App">
        <div>
            <button onClick={() => setShowEvents(false)}>Hide Events</button>
        </div>

        <div>
            <button onClick={() => setShowEvents(true)}>Show Events</button>
        </div>
        {showEvents && events.map((event, index) => (
            <div key={event.id}>
            <h2>{index} - {event.title}</h2>
            <button onClick={() => handleClick(event.id)}>delete event</button>
            </div>
        ))}
    </div>
  );
}
```



We are gonna show the list only when the showEvents state is true and hide when it is false

Now it would be bettter if we show hide event button when the list is visible and show button only when the list is hidden. For this we use conditionals in the buttons div tag

```jsx
return (
  <div className="App">
    {showEvents && (
      <div>
        <button onClick={() => setShowEvents(false)}>Hide Events</button>
      </div>
    )}
    {!showEvents && (
      <div>
        <button onClick={() => setShowEvents(true)}>Show Events</button>
      </div>
    )}
    {showEvents && events.map((event, index) => (
      <div key={event.id}>
        <h2>{index} - {event.title}</h2>
        <button onClick={() => handleClick(event.id)}>delete event</button>
      </div>
    ))}
  </div>
);
```



useState Limitations/Rules

1. useState or any other react hook must be called only at the top most level inside a react component .

```
function App() {
  const [showEvents, setShowEvents] = useState(true)
  const [events, setEvents] = useState([
    {title: "mario's birthday bash", id: 1},
    {title: "bowser's live stream", id: 2},
    {title: "race on moo moo farm", id: 3}
  ])

  console.log(showEvents)

  const handleClick = (id) => {
    setEvents((prevEvents) => {
      return prevEvents.filter((event) => {
        return id !== event.id
      })
    })
    useState() // not allowed
    console.log(id)
  }
```

2. A react hook has to be used within the scope of the react component, we cannot use hooks outside the compnent

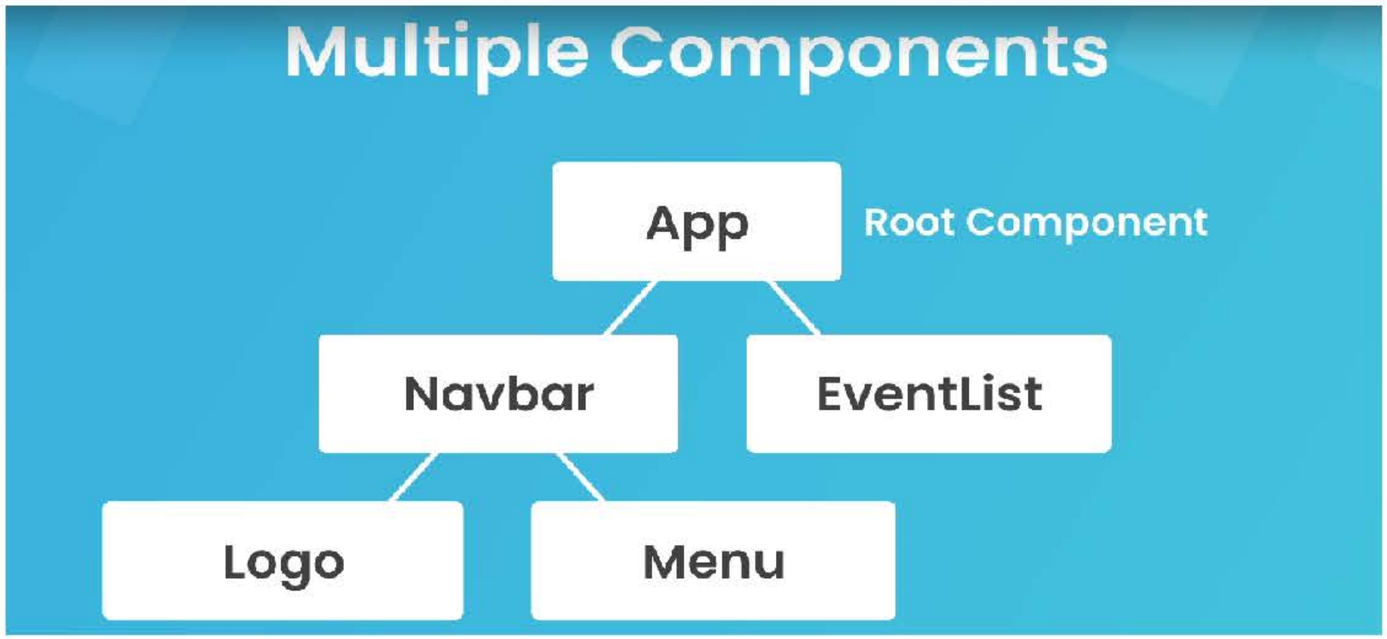
```
useState() // not allowed

function App() {
  const [showEvents, setShowEvents] = useState(true)
  const [events, setEvents] = useState([
    {title: "mario's birthday bash", id: 1},
    {title: "bowser's live stream", id: 2},
    {title: "race on moo moo farm", id: 3}
  ])

  console.log(showEvents)

  const handleClick = (id) => {
    setEvents((prevEvents) => {
      return prevEvents.filter((event) => {
        return id !== event.id
      })
    })
    console.log(id)
  }
```

Using Multiple Components

Till now, we worked with a single Component but website would typically be made of multiple components where each component would be responsible for a specific section of the site .

For example : a navbar component, a sidebar component, the middle section component etc

We work with multiple components by nesting the inside the root component. By doing so, we form a component tree in diagram form



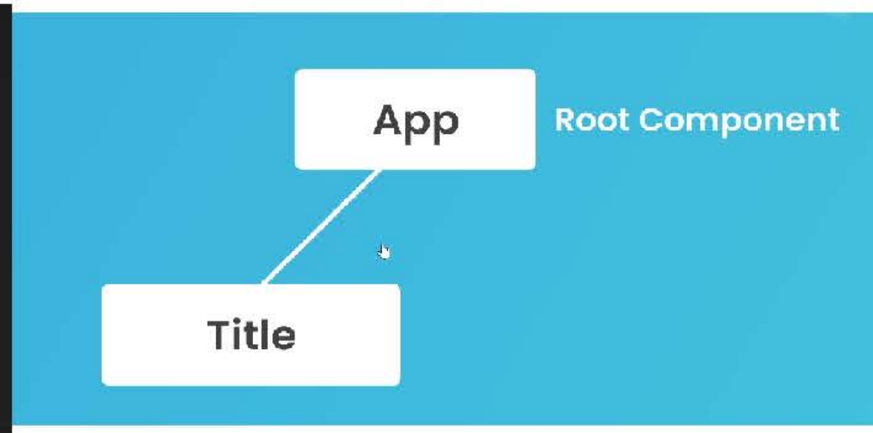
Each component has its own files, component fuinction etc

Creating a new component and nesting it in App component

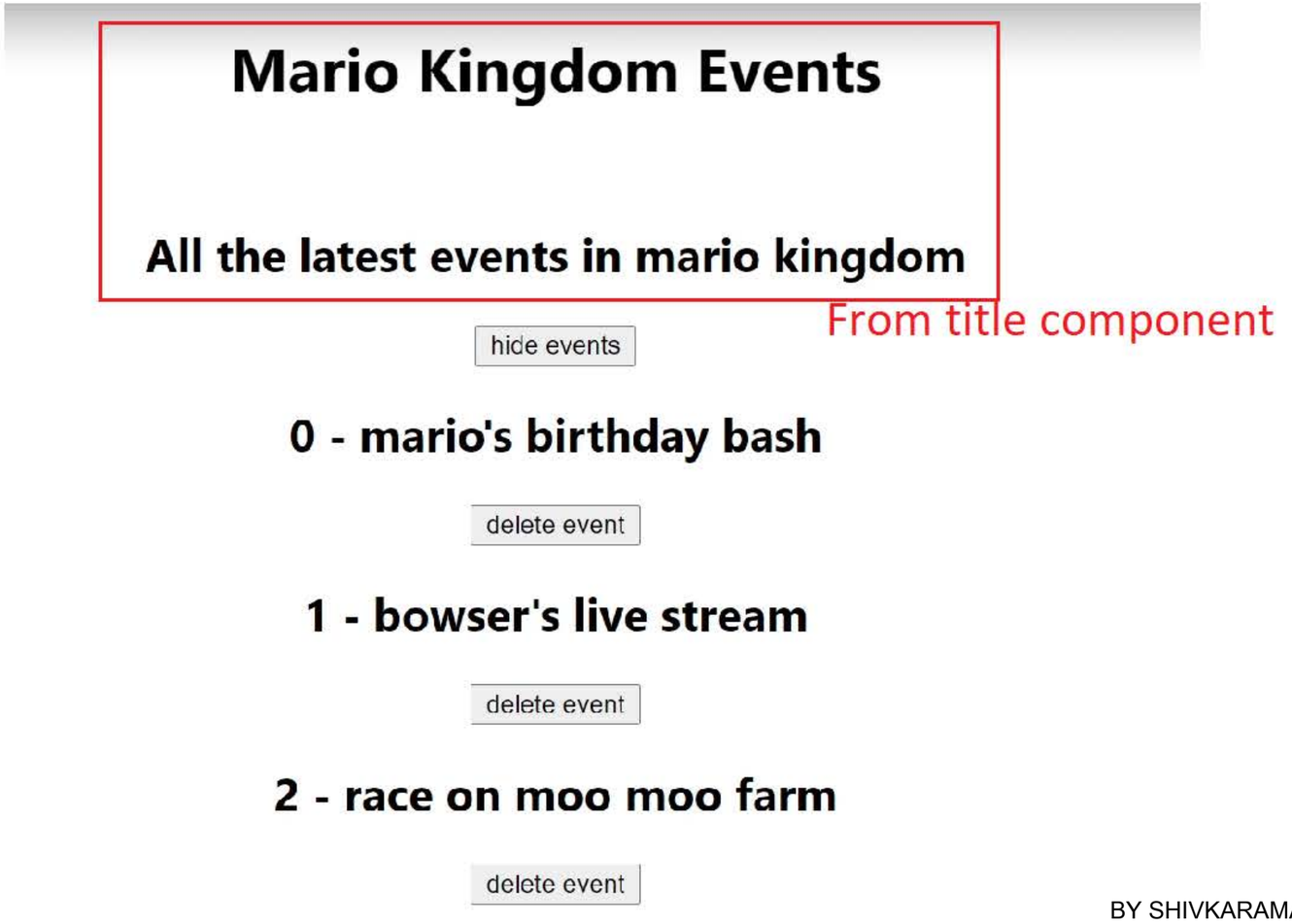
src > components > Title.js > Title

```jsx
export default function Title() {
  return (
    <div>
      <h1 className="title">Mario Kingdom Events</h1>
      <br />
      <h2 className="subtitle">All the latest events in mario kingdom</h2>
    </div>
  )
}
```

TITLE COMPONENT



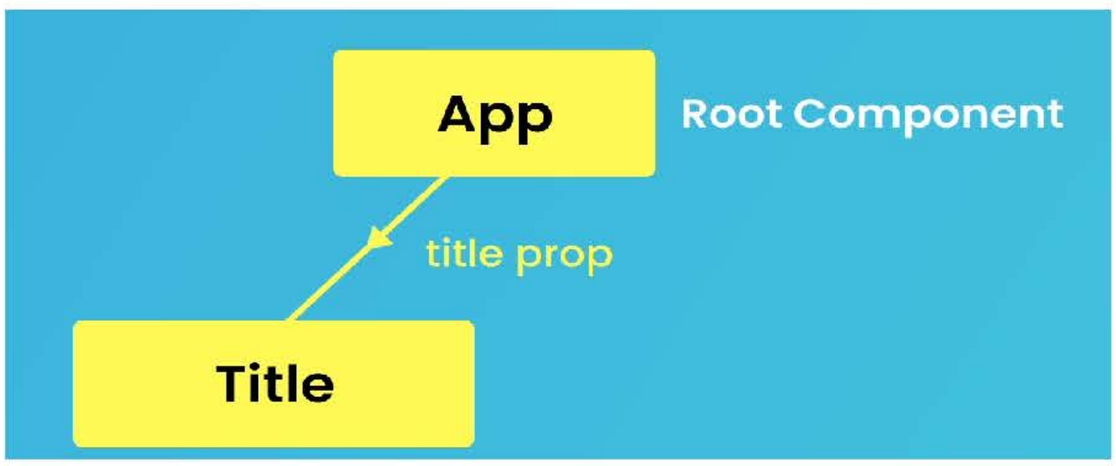

```jsx
import './App.css'
import { useState } from 'react'
import Title from './components/Title'

return (
  <div className="App">
    <Title />

    {showEvents && (
      <div>
        <button onClick={() => setShowEvents(false)}>Hide Events</button>
      </div>
    )}
    {!showEvents && (
      <div>
        <button onClick={() => setShowEvents(true)}>Show Events</button>
      </div>
    )}
    {showEvents && events.map((event, index) => (
      <div key={event.id}>
        <h2>{index} - {event.title}</h2>
        <button onClick={() => handleClick(event.id)}>delete event</button>
      </div>
    ))}
  </div>
);
```



BY SHIVKARAMAN

Props

Props is a way in react to pass data from a parent component to child component . props stands for properties.



The way you pass a prop to a component is just by making something like an attribute on the custom component

```
<Title title="Events in Your Area" />
```

When we pass a prop into a component, that component auotmatically recieves a prop object as a parameter in the function. It contains any prop that is sent to it

components > Title.js > Title

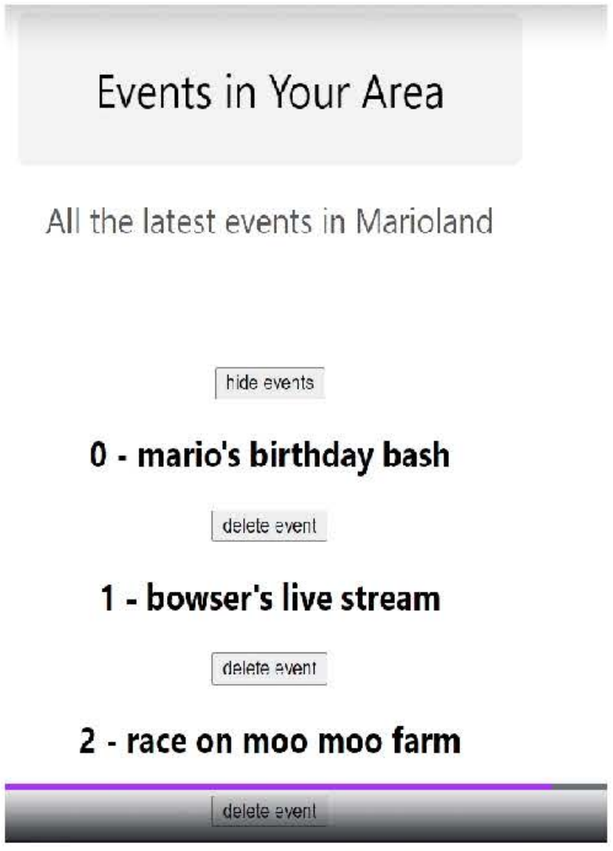
```
1  export default function Title(props) {
2    return (
3      <div>
4        <h1 className="title">{props.title}</h1>
5        <br />
6        <h2 className="subtitle">All the latest events in mario kingdom</h2>
7      </div>
8    )
9  }
```

You can pass multiple props at a time as well

```
const subtitle = "All the latest events in Marioland"
```

```
<Title title="Events in Your Area" subtitle={subtitle} />
```

```
export default function Title(props) {
  return (
    <div>
      <h1 className="title">{props.title}</h1>
      <br />
      <h2 className="subtitle">{props.subtitle}</h2>
    </div>
  )
}
```



Instead of using props.title , props.subtitle .... again and again for diferent props, we can destructure it

```
export default function Title({ title, subtitle }) {
  return (
    <div>
      <h1 className="title">{title}</h1>
      <br />
      <h2 className="subtitle">{subtitle}</h2>
    </div>
  )
}
```

React Fragments

WKT a react fragment can have atmost one root element. Sometimes this leads unnecessary commponents just to satisfy this condition

```jsx
export default function Title({ title, subtitle }) {
  return (
    <div>
      <h1 className="title">{title}</h1>
      <br />
      <h2 className="subtitle">{subtitle}</h2>
    </div>
  )
}
```

For example Say we needed only the contents withing the div tag but since there should be only one root element we have to enclose it within a div tag

To overcome this problem, we use react fragments

```jsx
export default function Title({ title, subtitle }) {
  return (
    <>
      <h1 className="title">{title}</h1>
      <br />
      <h2 className="subtitle">{subtitle}</h2>
    </>
  )
}
```

You can either use an

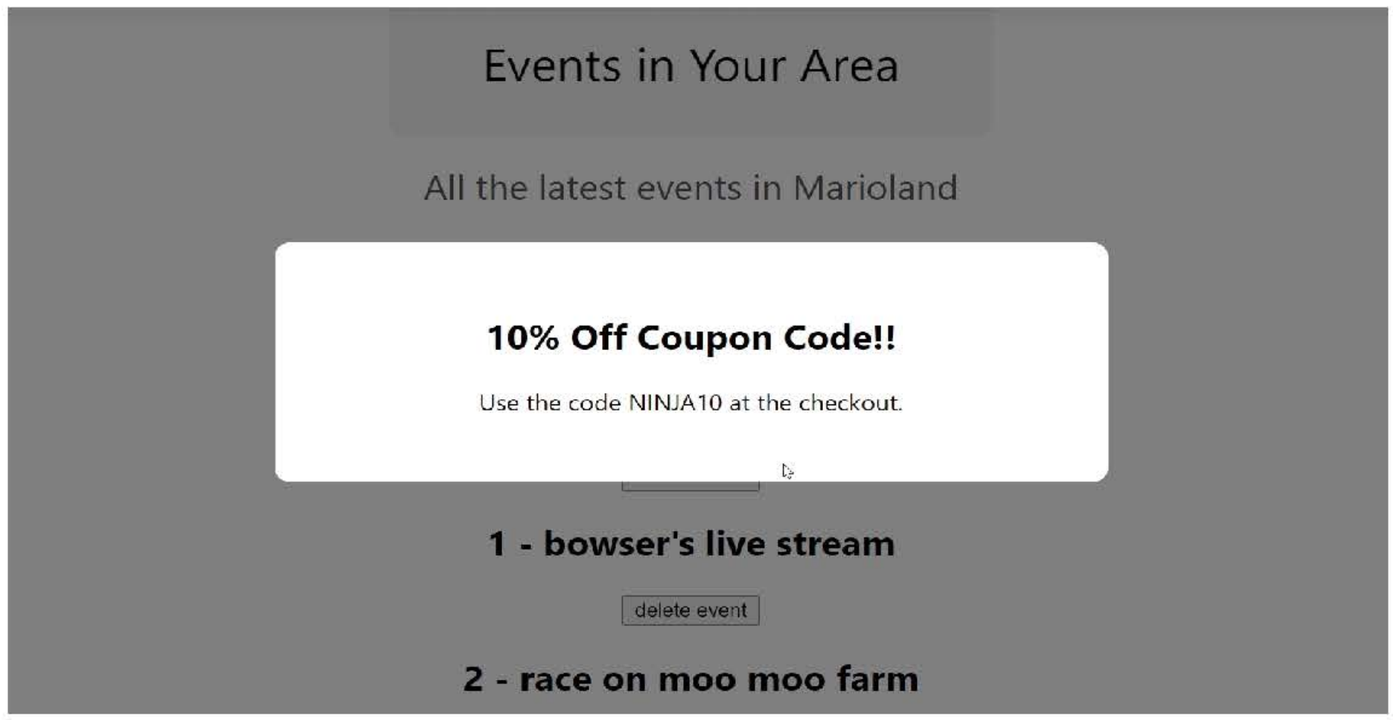
1. empty tag <> </> or
2. <React.fragment> </React.Fragment>

```jsx
<React.Fragment key={event.id}>
  <h2>{index} - {event.title}</h2>
  <button onClick={() => handleClick(event.id)}>delete event</button>
</React.Fragment>
```

When we are using props
we must use React.Fragment.
Using an empty tag will show error

Children Props (Making a Modal Component)

```
ponents > Modal.js > Modal
export default function Modal() {
  return (
    <div className="modal-backdrop">
      <div className="modal">
        <h2>10% Off Coupon Code!!</h2>
        <p>Use the code NINJA10 at the checkout.</p>
      </div>
    </div>
  )
}
```



We can use this modal function in App.js using by first importing **Modal** function from Modal.js and then using <Modal /> as we have seen befrore

But, What if we want to use this more reusable?
ie same modal style but different contents inside the contents -> WE USE CHILDREN PROPS (why not normal props? because they are used just to pass normal elements like string, bool etc)

We make the modal component more usable in the following way

```
export default function Modal(props) {
  return (
    <div className="modal-backdrop">
      <div className="modal">
        {props.children}
      </div>
    </div>
  )
}
```

```
<Modal>
  <h2>10% Off Coupon Code!!</h2>
  <p>Use the code NINJA10 at the checkout.</p>
</Modal>
```

children prop

⇩ we can use destructuring here as well

```
export default function Modal({ children }) {
  return (
    <div className="modal-backdrop">
      <div className="modal">
        {children}
      </div>
    </div>
  )
}
```

Function as Props

Till now we saw passing variables like string, boolean etc to another component.
But we can pass function as props as well.
We do this when the child component wants to manipulate state in the parent component

We created a modal in the previous page.
Obviously we dont want the modal to show all the time we sant to show it if some button is clicked or something like that

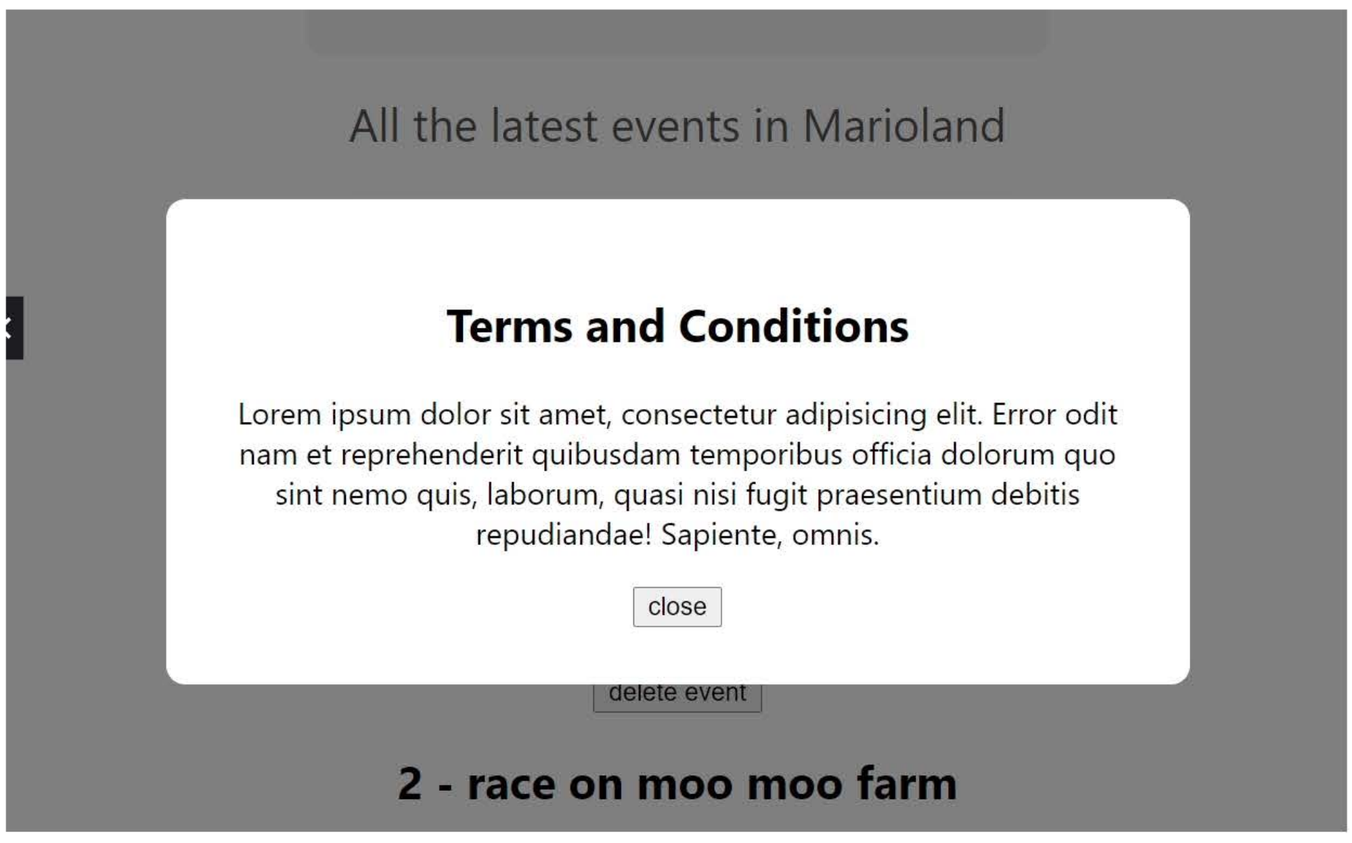
since the modal DOM(in App.js) and the modal component (modal.js) are in 2 different components and opening and closing of modal needs state manipulation and hence we use function as props for this purpose

App.js
(parent component)

```jsx
const [showModal, setShowModal] = useState(true)
const handleClose = () => {
  setShowModal(false)
}

{showModal && <Modal handleClose={handleClose}>
  <h2>Terms and Conditions</h2>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Error odit nam et
  reprehenderit quibusdam temporibus officia dolorum quo sint nemo quis, laborum,
  quasi nisi fugit praesentium debitis repudiandae! Sapiente, omnis.</p>
</Modal>}
```

Modal.js
(child component)

```jsx
export default function Modal({ children, handleClose }) {
  return (
    <div className="modal-backdrop">
      <div className="modal">
        {children}
        <button onClick={handleClose}>close</button>
      </div>
    </div>
  )
}
```



Modal closes when we press the close button

React Portal

A portal in react is to take a component and render it somewhere else in the DOM usually outside its scope of its parent component

Usually we nest the react componts inside the App component. But typically when we have modules on the website we might want to output it somewhere else on the website

```jsx
import ReactDOM from 'react-dom'
import './Modal.css'

export default function Modal({ children, handleClose }) {
  return ReactDOM.createPortal((
    <div className="modal-backdrop">
      <div className="modal">
        {children}
        <button onClick={handleClose}>close</button>
      </div>
    </div>
  ), document.body)
}
```

places the jsx template at the bottom of the body tag

syntax : ReactDom.createPortal('JSX template', 'whare you want to place the template')

Class Component

Class components are older ways of making components. But lets go through it in short.

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      date: new Date()
    }
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.props.name}</h1>
        <h2>It is {this.state.date.
        toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Class component

Creating useState

Crerating component

```
import { useState } from "react"

export default function Clock({ name }) {
  const [date, setDate] = useState(new Date())

  return (
    <div>
      <h1>Hello, {name}</h1>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  )
}
```

Function component

1. A class component is a JS class that inherits properties from React.component

2. Adding component -> We use render method inside the class and we return it

3. using props -> usnig this.props.prop_name

4. Creating useState -> Using constructor as shown in fig

1. A function component is just a plain function

2. We just return a JSX template

3. we just use the prop_name

4. Just using the useState

Styling react Applications

As seen earlier, we have used the global style sheet (index.css) and the component based style sheet (App.js ,..) to style the react application, but that is not the end of it

* index.css -> Styles that are side-wide and target general elements like div, p, button etc
* component based -> Styles that are related to the component

COMPONENT SPECIFIC STYLE SHEETS ARE ALSO GLOBAL

By that we mean that, all the component css files will be in the head region of the html like index.css file and these files are not scoped for that component alone

eg; If i add some style for h3 in modal.css, all the h3's will be affected globally

Inline Styles

We can style an element in the react component by using JS object of key value pair

key - property name

value - property value

```
style={{border: "4px solid", borderColor: "#ff4500"}}
```

Dynamic Inline Styles

```
export default function Modal({ children, handleClose, isSalesModal }) {
  return ReactDOM.createPortal((
    <div className="modal-backdrop">
      <div className="modal" style={{
        border: "4px solid",
        borderColor: isSalesModal ? "#ff4500" : "#555",
        textAlign: "center"
      }}>
        {children}
        <button onClick={handleClose}>close</button>
      </div>
    </div>
  ), document.body)
}
```

boolean value

Dynamic class

```
<button
  onClick={handleClose}
  className={isSalesModal ? "sales-btn" : ""}
>close</button>
```

CSS Modules

This is just another way of styling react components but with slight difference
CSS modules automatically scope their styles only to a specific component unlike the component based style sheets


```
import styles from './EventList.module.css'
```

inside → EventList.js

This is just a name given to the module and it can be any name but most of the developers use styles as name

```
<div className={styles.card} key={event.id}>
  <h2>{index} - {event.title}</h2>
  <button onClick={() => handleClick(event.id)}>delete event</button>
</div>
```

components > EventList.module.css > .card

```
.card {
  border: 1px solid #eee;
  box-shadow: 4px 4px 5px rgba(0,0,0,0.05);
  padding: 10px;
  max-width: 400px;
  margin: 20px auto;
  border-radius: 4px;
}
```

```
▶ <div class="EventList_card__3EJJ5">…</div>

.EventList_card__3EJJ5 {
  border: 1px solid #eee;
  box-shadow: 4px 4px 5px rgba(0,0,0,0.05);
  padding: 10px;
  max-width: 400px;
  margin: 20px auto;
  border-radius: 4px;
}
```

In browser

styles mentioned int the css module will be added to only those tags which use class name as mentioned above

There is a small catch here, the scoping of the css stuyles to only component only when we use class or id to identify the tag. If we use tags like button, div etc, it will be globally scoped like component based style sheet

For it to work, we need to give the tag a class or put its parents class in front of it

```
button {
  background-color: #555;
}
```

⇨

```
.card button {
  background-color: #555;
}
```

Whenever we have a hiphen (-) for class names we use a different syntax in react `className={styles['login-form']}`

Forms and Lables In React

In normal html form, this is the syntax we would use for input field and a lable

```
<label for="title"></label>
<input type="text" id="title" />
```

But in js, since for is a keyword used for 'for-loop', we use htmlFor instead of it

```
<label htmlFor="title">Event Title:</label>
<input type="text" id="title" />
```

We can use another format as well by enclosing it inside a label tag and using span tag for the text of label

```
<label>
  <span>Event Title:</span>
  <input type="text" />
</label>
```

Here is a react component for a form

```
export default function NewEventForm() {
  return (
    <form className="new-event-form">
      <label>
        <span>Event Title:</span>
        <input type="text" />
      </label>
      <label>
        <span>Event Date:</span>
        <input type="date" />
      </label>
      <button>Submit</button>
    </form>
  )
}
```

onChange() event

```
<input type="text" onChange={handleChange} />
```

handleChange function will be fired whenever user types something (similar to keyup event in JS)

```
export default function NewEventForm() {
  const [title, setTitle] = useState('')
  const [date, setDate] = useState('')

  const handleChange = (e) => {
    setTitle(e.target.value)
  }

  return (
    <form className="new-event-form">
      <label>
        <span>Event Title:</span>
        <input type="text" onChange={handleChange} />
      </label>
      <label>
        <span>Event Date:</span>
        <input type="date" onChange={handleChange} />
      </label>
      <button>Submit</button>
    </form>
  )
}
```

Using Inline function

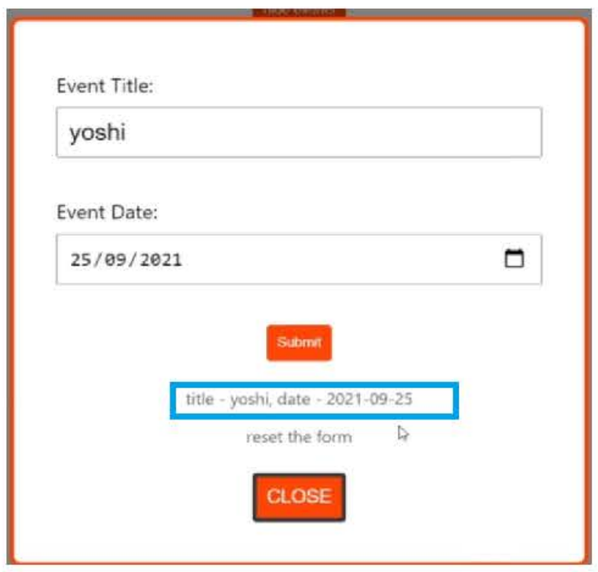
```
<label>
  <span>Event Title:</span>
  <input type="text" onChange={(e) => setTitle(e.target.value)} />
</label>
<label>
  <span>Event Date:</span>
  <input type="date" onChange={(e) => setDate(e.target.value)} />
</label>
```

Controled Input

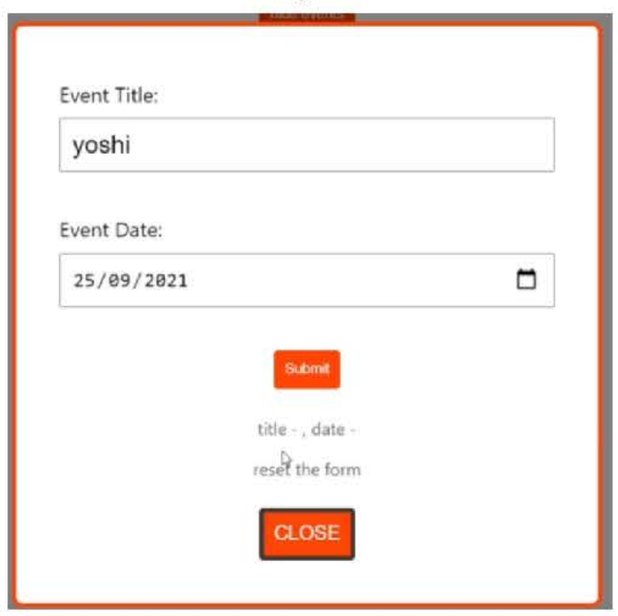
In the form we created, the data only bound between input and state only in one direction. By that we mean that when the input value changes that updates the state. , but whenever we change the state in some other function then that state update wouldnt be affected in the input field

To demonstrate this, let us create a function that resets the input field

```
const resetForm = () => {
  setTitle('')
  setDate('')
}

return (
  <form className="new-event-form">
    <label>
      <span>Event Title:</span>
      <input type="text" onChange={(e) => setTitle(e.target.value)} />
    </label>
    <label>
      <span>Event Date:</span>
      <input type="date" onChange={(e) => setDate(e.target.value)} />
    </label>
    <button>Submit</button>
    <p>title - {title}, date - {date}</p>
    <p onClick={resetForm}>reset the form</p>
  </form>
)
```



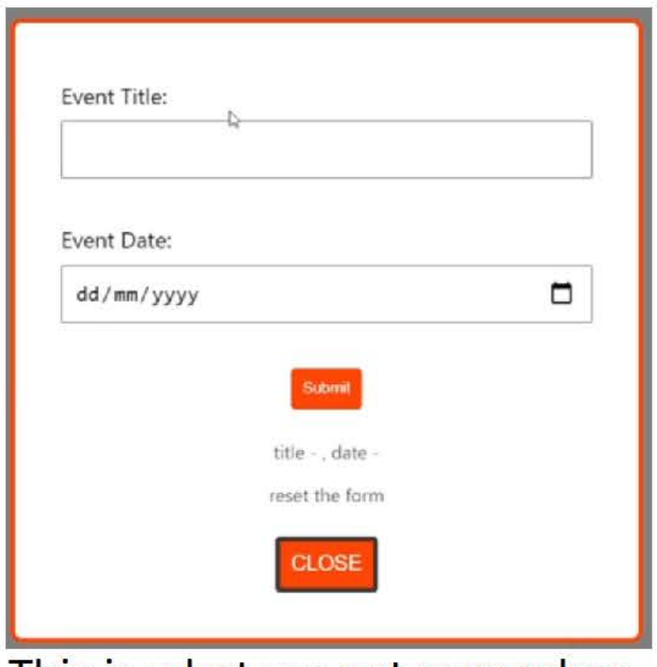
On clicking **reset** button



We may observe that on clicking the reset button, the states were reset back to the empty string as shown in the funnction, but the input fields didnt change.

So the way we setup the input connection bestween the input and the state is by adding the **value property** in the **input tag**

```
<label>
  <span>Event Title:</span>
  <input
    type="text"
    onChange={(e) => setTitle(e.target.value)}
    value={title}
  />
</label>
<label>
  <span>Event Date:</span>
  <input
    type="date"
    onChange={(e) => setDate(e.target.value)}
    value={date}
  />
</label>
```



This is what we get now when we click the reset button

Here we are controllin the value of the input from outside of the input (from reset func) and this is called as the **controlled input**

Submitting form

We can submit form by using **onSubmit** event in form field

```
<form className="new-event-form" onSubmit={handleSubmit}>
```

handleSubmit function is invoked when the form is submitted

```
const handleSubmit = (e) => {
  e.preventDefault()

  const event = {
    title: title,
    date: date,
    id: Math.floor(Math.random() * 10000)
  }
  console.log(event)
  resetForm()
}
```

The function just logs the listed property on the console and restes the form

useRef() hook

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

Syntax

```
const refContainer = useRef(initialValue);
```

So far we have used combination of useState() hook and onChange fucntion to keep track of the state and the user input so that we keep the input in sync with the actual values of the state

We have another way we can work using the refs
refs are a way we can get access to raw DOM elements directly. We can get reference to the DOM elements using the react hook **useRef**

```
const [title, setTitle] = useState('')
const [date, setDate] = useState('')

const resetForm = () => {
  setTitle('')
  setDate('')
}

const handleSubmit = (e) => {
  e.preventDefault()

  const event = {
    title: title,
    date: date,
    id: Math.floor(Math.random() * 10000)
  }

  addEvent(event)
  resetForm()
}
return (
  <form className="new-event-form" onSubmit={handleSubmit}>
    <label>
      <span>Event Title:</span>
      <input
        type="text"
        onChange={(e) => setTitle(e.target.value)}
        value={title}
      />
    </label>
    <label>
      <span>Event Date:</span>
      <input
        type="date"
        onChange={(e) => setDate(e.target.value)}
        value={date}
      />
    </label>
    <button>Submit</button>
  </form>
```

```
const title = useRef()
const date = useRef()

const resetForm = () => {
  title.current.value = ""
  date.current.value = ""
}

const handleSubmit = (e) => {
  e.preventDefault()
  console.log(title, date)

  const event = {
    title: title.current.value,
    date: date.current.value,
    id: Math.floor(Math.random() * 10000)
  }

  addEvent(event)
  resetForm()
}

return (
  <form className="new-event-form" onSubmit={handleSubmit}>
    <label>
      <span>Event Title:</span>
      <input
        type="text"
        ref={title}
      />
    </label>
    <label>
      <span>Event Date:</span>
      <input
        type="date"
        ref={date}
      />
    </label>
    <button>Submit</button>
  </form>
)
```

By adding the ref attribute inside the input tag, we are associating the 'title' refContainer with this input field

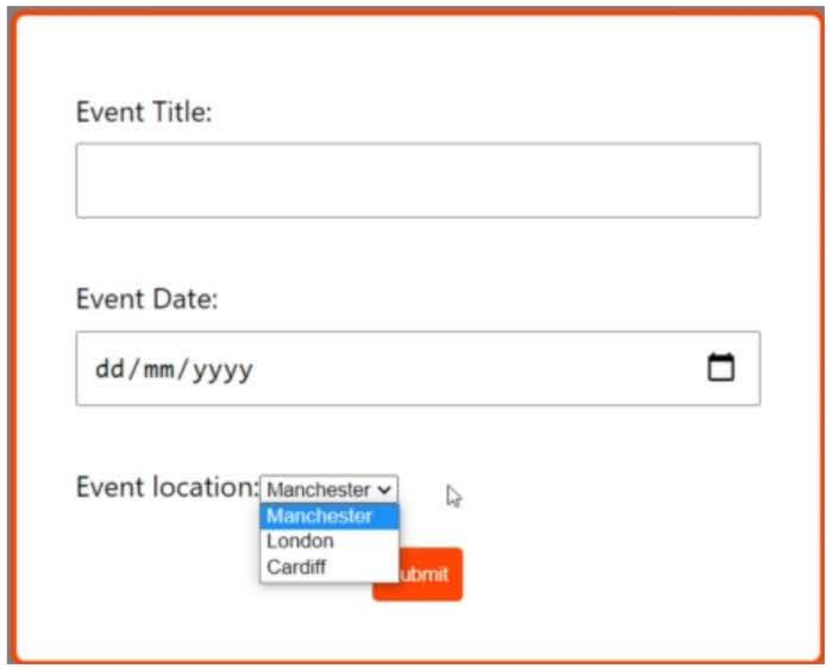
useState approach is a better and preferered approach

Select box

We add this to the form that we have created earlier

```
const [location, setLocation] = useState('manchester')
```

```
const resetForm = () => {
  setTitle('')
  setDate('')
  setLocation('manchester')
}

const handleSubmit = (e) => {
  e.preventDefault()

  const event = {
    title: title,
    date: date,
    location: location,
    id: Math.floor(Math.random() * 10000)
  }

  addEvent(event)
  resetForm()
}
```




```
<label>
  <span>Event location:</span>
  <select onChange={(e) => setLocation(e.target.value)}>
    <option value="manchester">Manchester</option>
    <option value="london">London</option>
    <option value="cardiff">Cardiff</option>
  </select>
</label>
```

Handling file input

```jsx
<label>
  <span>profile thumbnail:</span>
  <input
    required
    type="file"
    onChange={handleFileChange}
  />
</label>
```

```jsx
const [thumbnail, setThumbnail] = useState(null)
const [thumbnailError, setThumbnailError] = useState(null)

const handleFileChange = (e) => {
  setThumbnail(null)
  let selected = e.target.files[0]
  console.log(selected)

  if (!selected) {
    setThumbnailError('Please select a file')
    return
  }
  if (!selected.type.includes('image')) {
    setThumbnailError('Selected file must be an image')
    return
  }
  if (selected.size > 100000) {
    setThumbnailError('Image file size must be less than 100kb')
    return
  }

  setThumbnailError(null)
  setThumbnail(selected)
  console.log('thumbnail updated')

}
```

```
react_devtools_backend.js:4049
File {name: 'yoshi.png', lastModified: 1628435570615, lastMo
▼difiedDate: Sun Aug 08 2021 16:12:50 GMT+0100 (British Summe
r Time), webkitRelativePath: '', size: 7876, …}
  lastModified: 1628435570615
  ▶lastModifiedDate: Sun Aug 08 2021 16:12:50 GMT+0100 (Brit…
  name: "yoshi.png"
  size: 7876
  type: "image/png"
  webkitRelativePath: ""
  ▶[[Prototype]]: File
>
```

JSON SERVER

Instead of working with a database directly, we use a package called as the json-server which will alllow us to replicate the database using a Json file locally in this project. And it is gonna rap this json file with api endpoints that we can use to fetch and interactt with that data.
This is nothing but a pseudo-database


```
{
  "trips": [
    {
      "title": "2 Night Stay in Venice",
      "price": "£195"
    },
    {
      "title": "3 Night Stay in Paris",
      "price": "£295"
    },
    {
      "title": "4 Night Stay in London",
      "price": "£345"
    },
    {
      "title": "3 Night Stay in New York",
      "price": "£325"
    }
  ]
}
```



```
json-server --watch .\my_first_react_app\data\db.json
```

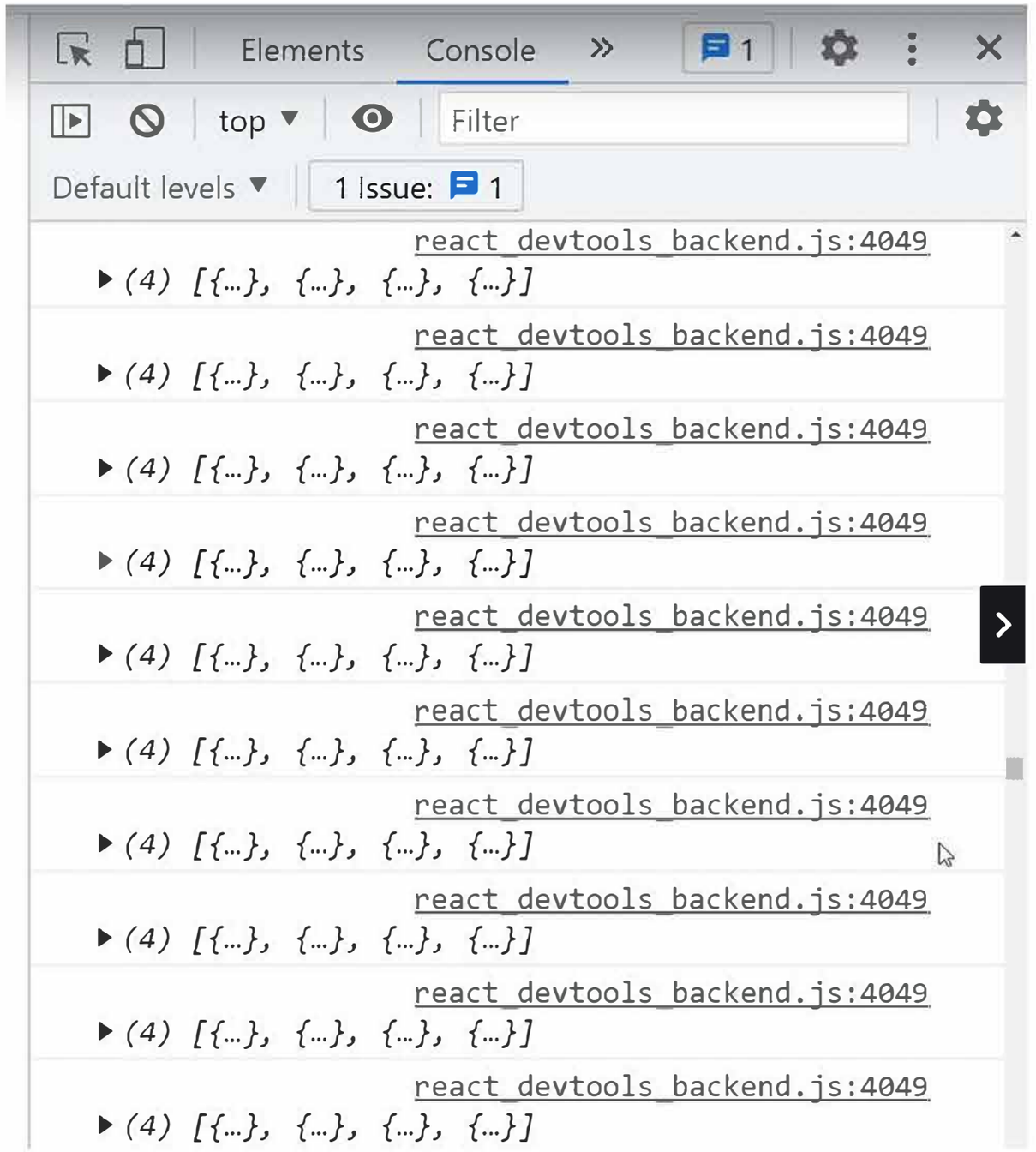
watches the file for changes

```
\{^_^}/ hi!

Loading .\my_first_react_app\data\db.json
Done

Resources
http://localhost:3000/trips

Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
Watching...
```

Why use useEffect hook?

src > components > TripList.js > TripList

```
1  import { useState } from "react"
2
3  export default function TripList() {
4    const [trips, setTrips] = useState([])
5
6    console.log(trips)
7
8    fetch('http://localhost:3000/trips')
9      .then(response => response.json())
10     .then(json => setTrips(json))
11     fetches data from the API we have created
12   return (
13     <div>
14       <h2>Trip List</h2>
15     </div>
16   )
17 }
18
```



We can see here that infinite requests are made here. This is because the react component is revaluated whenevet there is a state change.
Since the **trips** is changed from empty to non-empty, the component is reevaluated.
But now, the fetch is called again because we have not put any constraints on it not to run for second time => INFINITE LOOP

Fetching data using useEffect

useEffect is a react hook which allows us to run specific code only a specific times when certain conditions are met

The useEffect Hook allows you to perform side effects in your components.
Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

For the first component evaluation, the function runs automatically. Thereaftet when the component reruns aftere a state update or something like that, the useEffect function runs only if there is some changes in the dependency array.

src > components > TripList.js > TripList

```
1  import { useState, useEffect } from "react"
2
3  export default function TripList() {
4    const [trips, setTrips] = useState([])
5
6    useEffect(() => {
7      fetch('http://localhost:3000/trips')
8        .then(response => response.json())
9        .then(json => setTrips(json))
10   }, [])
11
12   console.log(trips)
13
14   return (
15     <div>
16       <h2>Trip List</h2>
17       <ul></ul>
18     </div>
19   )
20 }
21
```

Whenever we want the function to be fired only on first component evaluation, we pass an empty-dependency array .

useEffect with dependencies

```jsx
export default function TripList() {
  const [trips, setTrips] = useState([])
  const [url, setUrl] = useState('http://localhost:3000/trips')

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

  console.log(trips)

  return (
    <div className="trip-list">
      <h2>Trip List</h2>
      <ul>
        {trips.map(trip => (
          <li key={trip.id}>
            <h3>{trip.title}</h3>
            <p>{trip.price}</p>
          </li>
        ))}
      </ul>
      <div className="filters">
        <button onClick={() => setUrl('http://localhost:3000/trips?loc=europe')}>
          European Trips
        </button>
        <button onClick={() => setUrl('http://localhost:3000/trips')}>
          All Trips
        </button>
      </div>
    </div>
  )
}
```

any variable, function or useState which is defined outside useEffect should be declared as a dependency

button that displays trips with loc = europe or america

```json
{
  "trips": [
    {
      "title": "2 Night Stay in Venice",
      "price": "£195",
      "id": "1",
      "loc": "europe"
    },
    {
      "title": "3 Night Stay in Paris",
      "price": "£295",
      "id": "2",
      "loc": "europe"
    },
    {
      "title": "4 Night Stay in London",
      "price": "£345",
      "id": "3",
      "loc": "europe"
    },
    "loc": "europe"
    },
    {
      "title": "3 Night Stay in New York",
      "price": "£325",
      "id": "4",
      "loc": "america"
    }
```

Trip List

2 Night Stay in Venice

£195

3 Night Stay in Paris

£295

4 Night Stay in London

£345

3 Night Stay in New York

£325

European Trips | All Trips

when user clicks Europian Trips => url changes => url changes => one of the dependency of useEffect changed =>useEffect reruns its function

useCallback For Function Dependency

We cannot place asynchronous function inside useEffect directly => We have to use function dependency to do so

```
useEffect( async () => {
  const response = await fetch(url)
  const json = await response.json()
  setTrips(json)
}, [url])
```

This is not allowed in useEffect

```
useEffect(() => {
  fetch(url)
    .then(response => response.json())
    .then(json => setTrips(json))
}, [url])
```

Which is why we used fetch instead of async and await

```
const fetchTrips = async () => {
  const response = await fetch(url)
  const json = await response.json()
  setTrips(json)
}

useEffect(() => {
  fetchTrips()
}, [url, fetchTrips])
```

So this would how it wil look when we use a function as a dependency

But we have a problem here, the **fetchTrips** function is called infinite times here.
Eventhough the values of the dependencies **url** and **fetchTrips** hasnt changed, why is the useEffect refiring?

This is because technically, whenever our component is revaluated any function or any other varable that we declare inside that component are recreated and stored elsewhere in memory .When react compares the old function and the new fuction, it is not comparing the function name or the content inside it instead it compares the functions reference in the **memory which has changed.**
Since we are comparing references, so everytime the component reruns, the useEffect runs again => Infite loop

useEffect doesnt compare the reference of useState variable which is why we didnt see any infinite loop in the previous example

For functions we use something else => We wrap the function inside the **useCallback** hook

```
const fetchTrips = useCallback(async () => {
  const response = await fetch(url)
  const json = await response.json()
  setTrips(json)
})
```

What we are doing here is we are passing the funcetion as an arguement for useCallback hook.
On every evaluatio of the react component that cast functio is not not being recreated and therefore it is not seen as changed by the useEffect hook.

Just like the useEffect function, the useCallback hook takes dependency array as a second parameter.
It reruns when one of its dependencies update.

```
const [trips, setTrips] = useState([])
const [url, setUrl] = useState('http://localhost:3000/trips')

const fetchTrips = useCallback(async () => {
  const response = await fetch(url)
  const json = await response.json()
  setTrips(json)
}, [url])

useEffect(() => {
  fetchTrips()
}, [fetchTrips])

console.log(trips)
```

Now see that we no longer need to add url as a deoendency in useEffect because it is already a dependency of useCallback hook. So whenever url changes the function is recreated which will itself trigger the useEffect to run

For variabes and arrays -> Use useState
For functions -> Use useCallback

Creating custom Fetch Hook

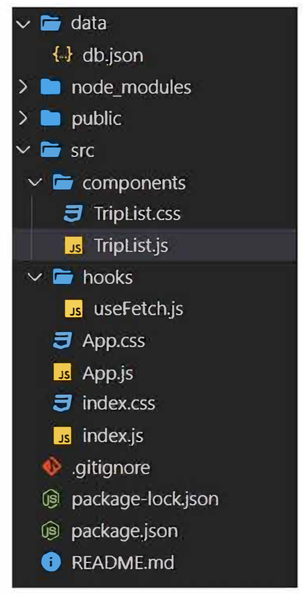
We just now saw how to fetch the data fron the API using useEffect and useCallback.
But what if we have to access the data from the api from diffferent components => the same code has to be used again

So instead of copying the same code in different component which would lead to redundancy, we create a custom fetch hook

```
const [trips, setTrips] = useState([])
const [url, setUrl] = useState('http://localhost:3000/trips')

const fetchTrips = useCallback(async () => {
  const response = await fetch(url)
  const json = await response.json()
  setTrips(json)
}, [url])

useEffect(() => {
  fetchTrips()
}, [fetchTrips])
```

src > hooks > useFetch.js > useFetch > useEffect() callback > fetchData

```
1  import { useState, useEffect } from 'react'
2
3  export const useFetch = (url) => {
4    const [data, setData] = useState(null)
5
6    useEffect(() => {
7      const fetchData = async () => {
8        const res = await fetch(url)
9        const json = await res.json()
10
11       setData(json)
12     }
13
14     fetchData()
15   }, [url])
16
17   return { data }
18
19 }
```



We could have defined fetchData outside the useEffect and add it as a dependency .
But we have used an alternatinve approach in which we define the function inside useEffect itself wehich will prevent it from being added as a dependency

```
import { useState, useEffect, useCallback } from "react"
import { useFetch } from '../hooks/useFetch'

// styles
import './TripList.css'

export default function TripList() {
  const [url, setUrl] = useState('http://localhost:3000/trips')
  const { data: trips } = useFetch(url)

  return (
    <div className="trip-list">
      <h2>Trip List</h2>
      <ul>
        {trips && trips.map(trip => (
          <li key={trip.id}>
            <h3>{trip.title}</h3>
            <p>{trip.price}</p>
          </li>
        ))}
      </ul>
      <div className="filters">
        <button onClick={() => setUrl('http://localhost:3000/trips?loc=europe')}>
          European Trips
        </button>
        <button onClick={() => setUrl('http://localhost:3000/trips')}>
          All Trips
        </button>
      </div>
```

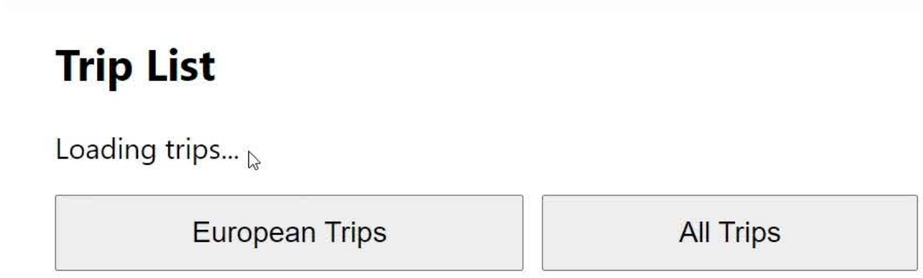
We have used **trips &&** here because, initially the value of data in custom hook is null and it will return null unitl the value is fetched from the api
So we make sure that this block of code runs only when data which is not null is returned form hook

Adding a Loading/Pending State

```jsx
import { useState, useEffect } from 'react'

export const useFetch = (url) => {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)

  useEffect(() => {
    const fetchData = async () => {
      setIsPending(true)

      const res = await fetch(url)
      const json = await res.json()

      setIsPending(false)
      setData(json)
    }

    fetchData()
  }, [url])

  return { data, isPening }
}
```

```jsx
export default function TripList() {
  const [url, setUrl] = useState('http://localhost:3000/trips')
  const { data: trips, isPending } = useFetch(url)

  return (
    <div className="trip-list">
      <h2>Trip List</h2>
      {isPending && <div>Loading trips...</div>}
      <ul>
        {trips && trips.map(trip => (
          <li key={trip.id}>
            <h3>{trip.title}</h3>
            <p>{trip.price}</p>
          </li>
        ))}
      </ul>
      <div className="filters">
        <button onClick={() => setUrl('http://localhost:3000/trips?loc=europe')}>
          European Trips
        </button>
        <button onClick={() => setUrl('http://localhost:3000/trips')}>
          All Trips
        </button>
      </div>
```

This will display loading message until data is fetched



Handling Errors

```jsx
import { useState, useEffect } from 'react'

export const useFetch = (url) => {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const [error, setError] = useState(null)

  useEffect(() => {
    const fetchData = async () => {
      setIsPending(true)

      try {
        const res = await fetch(url)
        if (!res.ok) {
          throw new Error(res.statusText)
        }
        const json = await res.json()

        setIsPending(false)
        setData(json)
        setError(null)
      } catch (err) {
        setIsPending(false)
        setError('Could not fetch the data')
        console.log(err.message)
      }

    }

    fetchData()
  }, [url])

  return { data, isPending, error }
}
```

We introduce a new state called as error
We use a try - catch block to throw error if any

As we have already seen in JS that the await only throws network error we need to separately monitor other errors

So we use the resopnse object if the data was successfuly fetched or not.

```
▼ Response
    body: (...)
    bodyUsed: true
  ▶ headers: Headers {}
    ok: false
    redirected: false
    status: 404
    statusText: "Not Found"
    type: "cors"
    url: "http://localhost:3000/tripsdas"
  ▶ [[Prototype]]: Response
```

This is true only when data is fetched successfully

```jsx
<div className="trip-list">
  <h2>Trip List</h2>
  {isPending && <div>Loading trips...</div>}
  {error && <div>{error}</div>}
  <ul>
    {trips && trips.map(trip => (
      <li key={trip.id}>
        <h3>{trip.title}</h3>
        <p>{trip.price}</p>
      </li>
    ))}
  </ul>
```

Trip List

Could not fetch the data

European Trips | All Trips

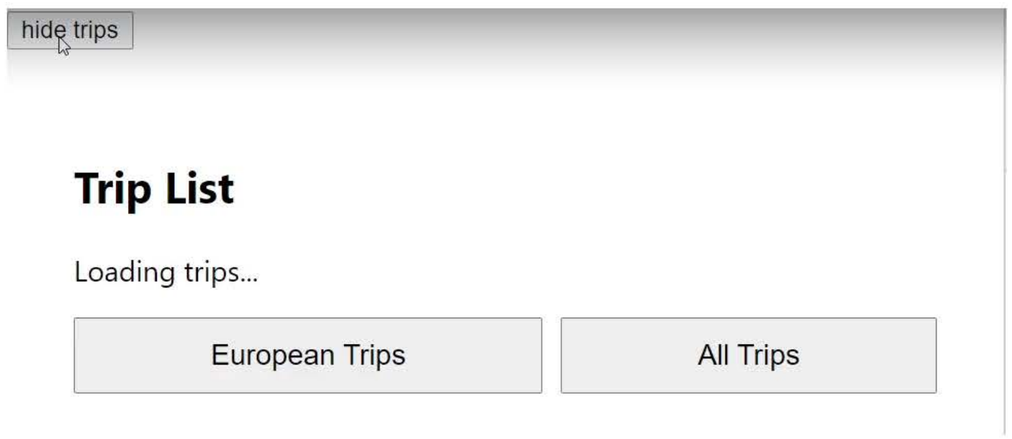
```
top ▼ | Filter | Default levels ▼
1 Issue: 1
[HMR] Waiting for update signal from WDS...                    log.js:24
▶ GET http://localhost:3000/tripsdas 404 (Not Found)           useFetch.js:13
Not Found                                                      react_devtools_backend.js:4049
>
```

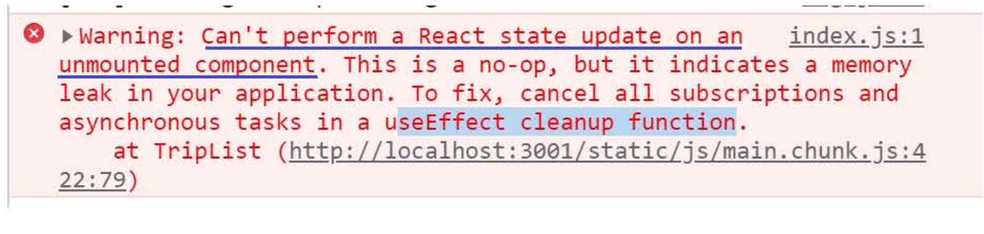
Why we need a cleanup Function?

```
src > App.js > App
1  import { useState } from 'react'
2  import './App.css';
3  import TripList from './components/TripList'
4
5  function App() {
6    const [showTrips, setShowTrips] = useState(true)
7
8    return (
9      <div className="App">
10       <button onClick={() => setShowTrips(false)}>hide trips</button>
11       {showTrips && <TripList />}
12     </div>
13   );
14 }
15
16 export default App;
17
```

We have introduces a new button **hide trips** which when clicked hides the triplist
So the trip list will be visible only when **showTrips** state is true
Initially wen the page is loaded, showTrips is true and the trip list will be loaded without any error and will be hidden when the button is clicked.

Now porblem is when we click the button while the fetch is going on





What is happending is when we load the page and the TripList component starts to load , it uses the useFetch hook and causes the fetch to start fetching the data.
In the meantime while the fetch is goingon in the background and we click the button the component is removed from the page. But when the fetch going on in background is complete , it is trying to update the state in that component which was just removed which is not possible ie trying to update the state of the component which doesnot exist

This is what the highlighted line in the above error means

So we need a function which will abort the fetch request -> Cleanup Function

Cleanup Function

Cleanup function is used to abort the fetch request if any component using the hook unmounts while the fetch is still going on in the background.
So cleanup functio is a funtion that we return inside a **useEffect** function and the cleanup function will always run when the component using this useEffect hook unmounts .

In our case, the tripList component is using the useFetch component which is indirectly using the useEffect hook as well. Meaning when the TripList component unmounts, it will fire the cleanup function .

We abort a fetch request using an abort controller

```
useEffect(() => {
  const controller = new AbortController()

  const fetchData = async () => {
    setIsPending(true)

    try {
      const res = await fetch(url, { signal: controller.signal })
      if (!res.ok) {
        throw new Error(res.statusText)
      }
      const json = await res.json()

      setIsPending(false)
      setData(json)
      setError(null)
    } catch (err) {
      if (err.name === "AbortError") {
        console.log('the fetch was aborted')
      } else {
        setIsPending(false)
        setError('Could not fetch the data')
      }
    }

  }

  fetchData()

  return () => {
    controller.abort()
  }      CLEANUP FUNCTION
}, [url])

  return { data, isPending, error }
}
```

Associating the abort controller with the fetch request so that the cleanup function knows which fetch request is to be aborted

Aborts the fetch request associated with the AbortController

It is alwaya a good practice to include a cleanup function when we are using useEffect and doing something asynchronous to fetch data

Making a POST Request

Till now we saw how we can fetch data from the JSON server ie **GET** method. Now we will see how to add new data to server using **POST** method

```
import { useState, useEffect } from 'react'

export const useFetch = (url) => {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const [error, setError] = useState(null)

  useEffect(() => {
    const controller = new AbortController()

    const fetchData = async () => {
      setIsPending(true)

      try {
        const res = await fetch(url, { signal: controller.signal })
        if (!res.ok) {
          throw new Error(res.statusText)
        }
        const json = await res.json()

        setIsPending(false)
        setData(json)
        setError(null)
      } catch (err) {
        if (err.name === "AbortError") {
          console.log('the fetch was aborted')
        } else {
          setIsPending(false)
          setError('Could not fetch the data')
        }
      }

    }

    fetchData()

    return () => {
      controller.abort()
    }
  }, [url])

  return { data, isPending, error }
}
```

useFetch hook for **GET methos ie to only fetch data**

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

export const useFetch = (url, method = "GET") => {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const [error, setError] = useState(null)
  const [options, setOptions] = useState(null)

  const postData = (postData) => {
    setOptions({
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify(postData)
    })
  }

  useEffect(() => {
    const controller = new AbortController()

    const fetchData = async (fetchOptions) => {
      setIsPending(true)

      try {
        const res = await fetch(url, { ...fetchOptions, signal: controller.signal })
        if(!res.ok) {
          throw new Error(res.statusText)
        }
        const data = await res.json()

        setIsPending(false)
        setData(data)
        setError(null)
      } catch (err) {
        if (err.name === "AbortError") {
          console.log("the fetch was aborted")
        } else {
          setIsPending(false)
          setError('Could not fetch the data')
        }
      }
    }

    if (method === "GET") {
      fetchData()
    }
    if (method === "POST" && options) {
      fetchData(options)
    }

    return () => {
      controller.abort()
    }

  }, [url, options, method])

  return { data, isPending, error, postData }
}
```

GET or **POST** request
It is GET by default

Data that we want to save in server

This otulines the type of data we are sending to the server ie JSON

Converts the data which is to be uploaded to JSON format

This is the function that will be called by the parent component when some data is to be uploaded

How does POST work?

We call the **useFetch** with the **POST** request
Then the **useEffect** hook is gonna fire the function and it is going to perform the check if the methos id **POST** which is true in this case.
Yes the method is **POST** but we dont have any **options** set yet. It is still null. So no fetch request is sent yet.

We return the **postData** function to the parent component which can be called by the parent component to upload the desired data.

When the parent component calls the **postData** function, it sets the options as shown above.

Right now we just changed the **options** state which is also a dependency of the **useEffect** hook. Hence the useEffect function runs again and now we have a value for options =>we invoke the **fetchData** function which will make the **POST** request for us.

```
const { postData, data, error } = useFetch('http://localhost:3000/recipes', 'POST')

postData({ title, ingredients, method, cookingTime: cookingTime + ' minutes' })
```

POST REQUEST FROM PARENT COMPONENT

Multi-Page React Sites

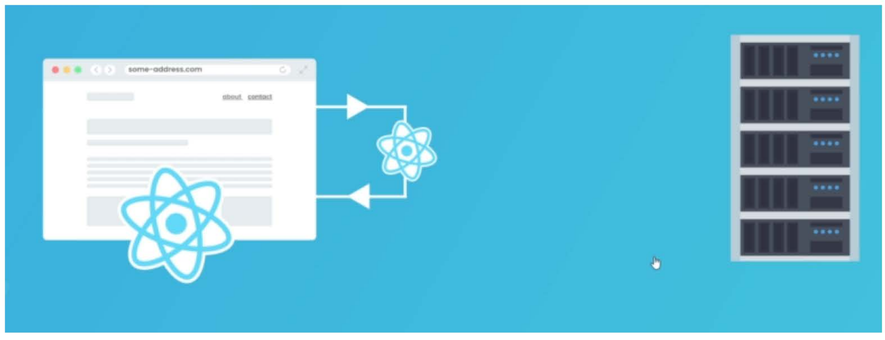
So far we have dealt with single page react components.

React-router is a JS package which we can install into a react project to impliment multi-page react applications.

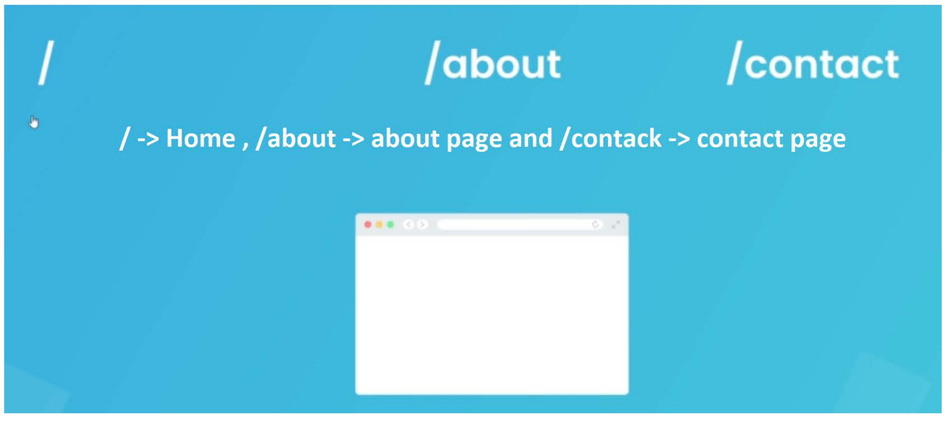
But remember, when we create a react site it still technically is a single reacct page that makes up the website

As we discussed earlier, when we talked about single page applications we said that the server sends back an initial response which contains an empty html page. Then react dynamically injects react components into the page.

But when we work with the react router we dont send an extra request to servere for new pages, but instead the **router intercepts those requests**, looks at the route we are heading to and swaps out the page with the current page. This makes the experiene of switching pages much quicker.



Say we have a page with 3 Routes OR 3 pages



To navigate between pages, we can create a react router so that
if we go to just '**/**' it injects a home component
If we go to **/about** it injects the about component and
if we go to **/contact** it injects the contact component

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.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Setup

`npm install react-router-dom` -Installs the latest version of react-router

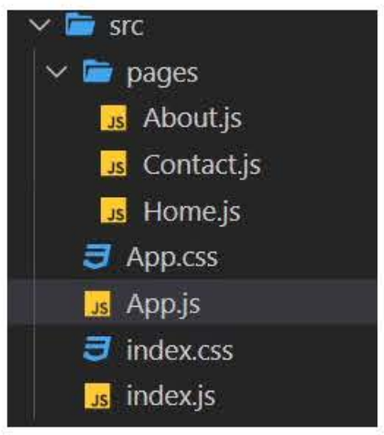
src > App.js > App

```
1  import './App.css'
2  import { BrowserRouter, Route } from 'react-router-dom'
3
4  // page components
5  import Home from './pages/Home'
6  import Contact from './pages/Contact'
7  import About from './pages/About'
8
9  function App() {
10   return (
11     <div className="App">
12       <BrowserRouter>
13         <nav>
14           <h1>My Articles</h1>
15         </nav>
16
17         <Route path="/">
18           <Home />
19         </Route>
20         <Route path="/about">
21           <About />
22         </Route>
23         <Route path="/contact">
24           <Contact />
25         </Route>
26
27       </BrowserRouter>
28     </div>
29   );
30 }
```

Used to make a route

It wraps our entire appliction that includes routes

It surrounds anyting that is gonna use the router



React contains three different packages for routing. These are:

1. **react-router:** It provides the core routing components and functions for the React Router applications.
2. **react-router-native:** It is used for mobile applications.
3. **react-router-dom:** It is used for web applications design.

There are two types of router components:

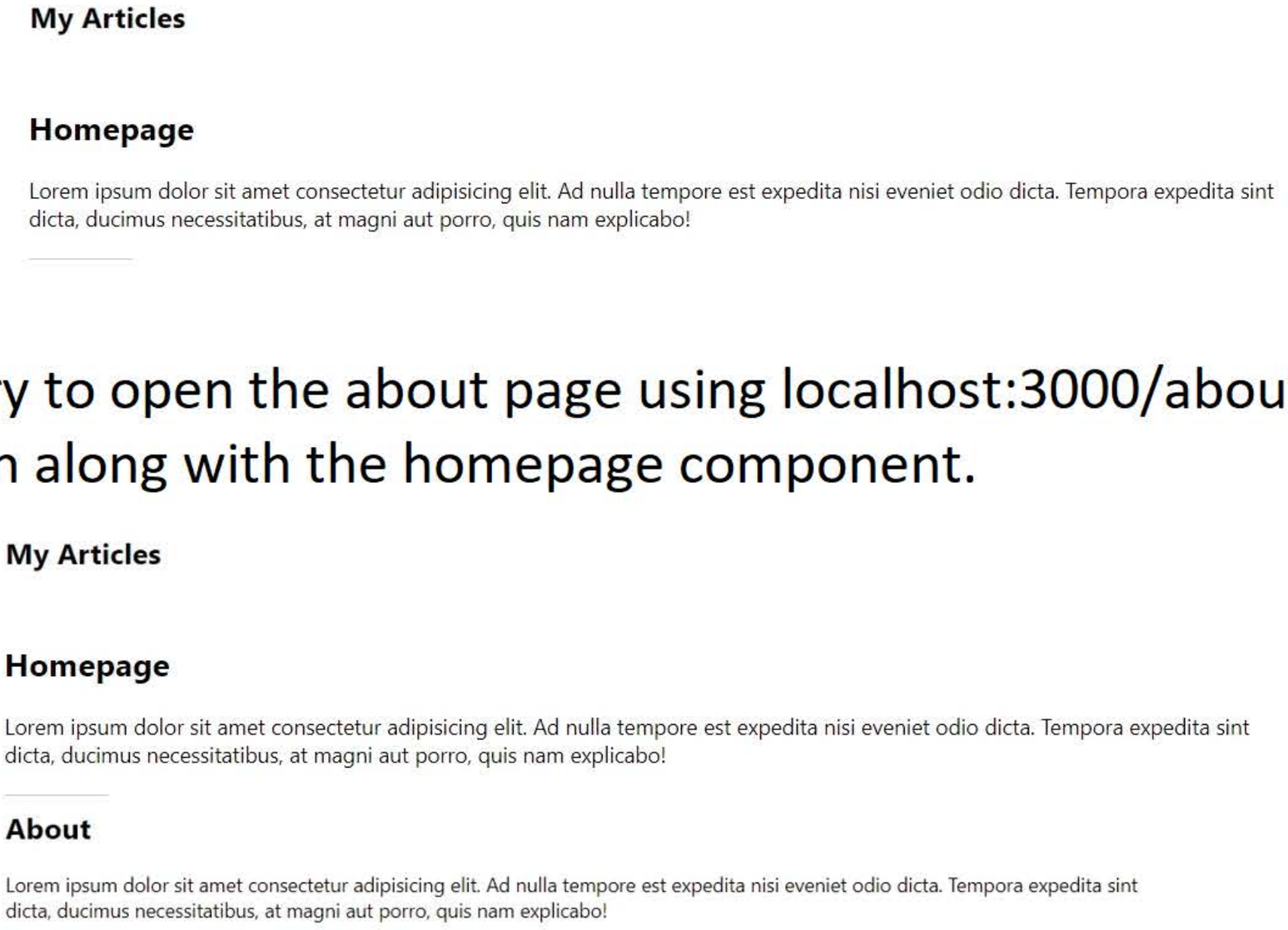
<BrowserRouter>: It is used for handling the dynamic URL.

<HashRouter>: It is used for handling the static request.

Switch And Exact Match

We have a small problem here.

When we load the home component using localhost:3000/, we get the home component like this



But when i try to open the about page using localhost:3000/about , I get the about section along with the homepage component.

This is the same for contact page as well

This is happening because when react router sees at the route on the search bar, it scans the router available in the **<Route>** section one by one and check if it exists in the URL that we have put in the browser, if yes it adds the contents of that component to the dom.

Eg : localhost:3000/about -> It first checks for **/** and checks if it is present as a path in Route section which is true. So it loads the content of the Home component. Then it checks for **about** and loads its content on the dom if true.

To overcome this problme, we use **Switch**

```
<div className="App">
  <BrowserRouter>
    <nav>
      <h1>My Articles</h1>
    </nav>

    <Switch>
      <Route exact path="/">
        <Home />
      </Route>
      <Route path="/about">
        <About />
      </Route>
      <Route path="/contact">
        <Contact />
      </Route>
    </Switch>

  </BrowserRouter>
</div>
```

What switch does is it allows only content of on of the routes on the dom ie either Homepage or About or Contact.

But we have another problem now. Any route we ge, it displays only Homepage component. Why? Because every path has URL beginning with '/' . So the homepage is loaded and react doesnt look for further routes as only one route is allowed at a time.

Which is why we use **exact** to check if the URL exactly matches the path and only load if true

Links and NavLinks

Including navigation links for different pages

```
<BrowserRouter>
  <nav>
    <h1>My Articles</h1>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/contact">Contact</a>
  </nav>

  <Switch>
    <Route exact path="/">
      <Home />
    </Route>
    <Route path="/about">
      <About />
    </Route>
    <Route path="/contact">
      <Contact />
    </Route>
  </Switch>

</BrowserRouter>
```

This works fine.But everytime we click the button, but the webpage is making a new request to the server and the server returns a blank html page which is filled by the react app.

This is not what we want.
We want the reacct router to intercept those reqest.

HOW DO WE DO THAT? -> Using a special component called as **Link component** instead of anchor tag

```
import { BrowserRouter, Route, Switch } from 'react-router-dom'
```

```
<nav>
  <h1>My Articles</h1>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  <Link to="/contact">Contact</Link>
</nav>
```

We can use **NavLink** instead of Link as well

```
import { BrowserRouter, Route, Switch, NavLink } from 'react-router-dom'
```

```
<nav>
  <h1>My Articles</h1>
  <NavLink to="/">Home</NavLink>
  <NavLink to="/about">About</NavLink>
  <NavLink to="/contact">Contact</NavLink>
</nav>
```

What is the difference between Link and NavLink?

The main difference is that NavLInk is used for navigations. What is does is it applies an **active class** to any link that is currently active.
Eg. When we click the about button, what the NavLink does is it checks if the /about is present in the URL and if so, it adds a **active** class to that tag so that we can style that tag differently

```
nav a.active {
  color: white;
  background: #333;
  text-decoration: none;
}
```

The same problem arises as befofe ie home page being active when we go to about or contact page. The reason and solution is also the same for this

```
<nav>
  <h1>My Articles</h1>
  <NavLink exact to="/">Home</NavLink>
  <NavLink to="/about">About</NavLink>
  <NavLink to="/contact">Contact</NavLink>
</nav>
```

Fetching Data

db.json

```
"articles": [
  {
    "id": "1",
    "title": "Welcome to the Site",
    "author": "Mario",
    "body": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Nisi, ullam eos
    dignissimos aperiam rerum qui suscipit cum nobis, totam ea tenetur perferendis
    praesentium corporis possimus ducimus et minima voluptatum. Numquam mollitia
    culpa consectetur unde illum est aut dicta eligendi vero molestias impedit sint,
    maiores saepe voluptas necessitatibus excepturi ducimus repudiandae, non quidem
    nobis veritatis! Libero neque, cumque illo est corrupti eaque recusandae ipsum,
    ut debitis vitae molestias deleniti voluptates distinctio sapiente autem. Tempore
    aperiam minima sit atque, tempora doloribus blanditiis id ipsum. Distinctio quos
    nisi, totam sunt ex voluptatum? Neque alias laborum ipsum doloremque fuga earum
    in autem. Hic alias omnis facilis facere eum assumenda deleniti ad, maiores
    laudantium temporibus odio non, molestiae dolorum! Quo mollitia ex sapiente
    maiores excepturi?"
  },
```

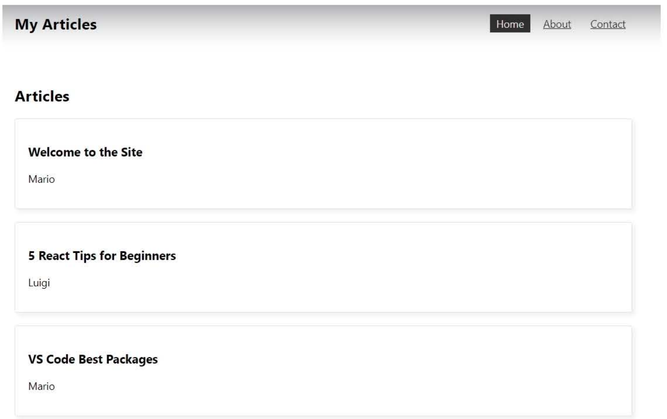
useFetch custom hook

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

export const useFetch = (url) => {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const [error, setError] = useState(null)

  useEffect(() => {
    const controller = new AbortController()

    const fetchData = async () => {
      setIsPending(true)

      try {
        const res = await fetch(url, { signal: controller.signal })
        if(!res.ok) {
          throw new Error(res.statusText)
        }
        const data = await res.json()

        setIsPending(false)
        setData(data)
        setError(null)
      } catch (err) {
        if (err.name === "AbortError") {
          console.log("the fetch was aborted")
        } else {
          setIsPending(false)
          setError('Could not fetch the data')
        }
      }
    }

    fetchData()

    return () => {
      controller.abort()
    }

  }, [url])

  return { data, isPending, error }
}
```

We will fetch the data fro the JSON server using the custom hook that we had created in the earlier lessons

App.js

```
import { useFetch } from '../hooks/useFetch'

export default function Home() {
  const { data: articles, isPending, error } = useFetch('http://localhost:3000/articles')

  return (
    <div className="home">
      <h2>Articles</h2>
      {isPending && <div>Loading...</div>}
      {error && <div>{error}</div>}
      {articles && articles.map((article) => (
        <div key={article.key} className="card">
          <h3>{article.title}</h3>
          <p>{article.author}</p>
        </div>
      ))}
    </div>
  )
}
```



We will show the body of the article on clicking the article using React Routes

Route Parameters

/articles/:id

/articles/2

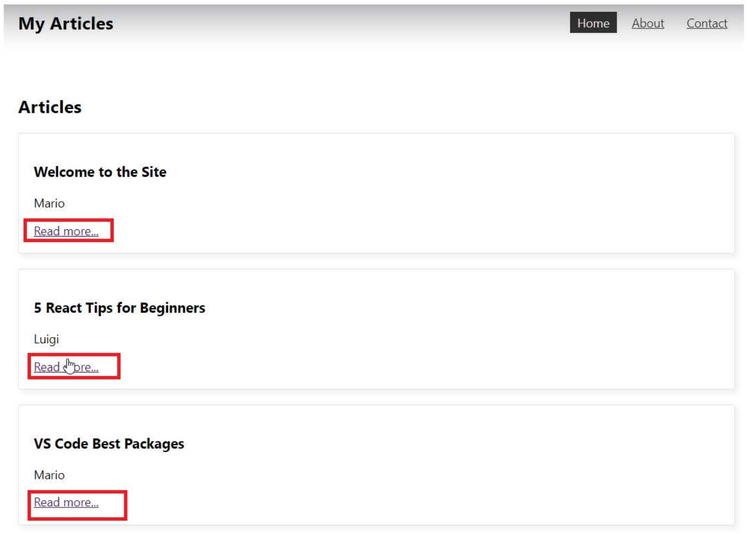
/articles/3

/articles/999

These changable parts of the route are called as route parameter

: -> Tells react that it is a route parameter

```
export default function Home() {
  const { data: articles, isPending, error } = useFetch('http://localhost:3000/articles')

  return (
    <div className="home">
      <h2>Articles</h2>
      {isPending && <div>Loading...</div>}
      {error && <div>{error}</div>}
      {articles && articles.map((article) =>  (
        <div key={article.key} className="card">
          <h3>{article.title}</h3>
          <p>{article.author}</p>
          <Link to={`/articles/${article.id}`}>Read more...</Link>
        </div>
      ))}
    </div>
  )
```



We have to create a route for the Link and a component to load for that link

```
    <nav>
      <h1>My Articles</h1>
      <NavLink exact to="/">Home</NavLink>
      <NavLink to="/about">About</NavLink>
      <NavLink to="/contact">Contact</NavLink>
    </nav>

    <Switch>
      <Route exact path="/">
        <Home />
      </Route>
      <Route path="/about">
        <About />
      </Route>
      <Route path="/contact">
        <Contact />
      </Route>
      <Route path="/articles/:id">
        <Article />
      </Route>
    </Switch>
```

src > pages > Article.js > Article

```
export default function Article() {
  return (
    <div>
      Article page
    </div>
  )
}
```

Article.js

localhost:3000/articles/1

When we click on **read more** of any article, this page is loaded

useParams hook

```jsx
import { useParams } from "react-router-dom"

export default function Article() {
  const params = useParams()

  return (
    <div>
      Article page - { params.id }
    </div>
  )
}
```

```jsx
<Route path="/articles/:id">
  <Article />
</Route>
```

⇩ Destructuring

```jsx
export default function Article() {
  const { id } = useParams()

  return (
    <div>
      Article page - { id }
    </div>
  )
}
```

The useParams() hook is a React Router hook that allows you to access the parameters of the current URL. This can be useful if you want to dynamically render content based on the URL parameters.

```jsx
export default function Article() {
  const { id } = useParams()
  const url = 'http://localhost:3000/articles/' + id
  const { data: article, isPending, error } = useFetch(url)

  return (
    <div>
      {isPening && <div>Loading...</div>}
      {error && <div>{error}</div>}
      {article && (
        <div>
          <h2>{article.title}</h2>
          <p>By {article.author}</p>
          <p>{article.body}</p>
        </div>
      )}
    </div>
  )
}
```

useParams hook is used to fetch data based on the parameters in the url

Programatic Redirect

When we try to go to an article that doesnot exist,we get this message

But it would be better if we could go back to the home page

To redirect to some page, we use the **useHistory** hook

```
import { useEffect } from "react"
import { useParams, useHistory } from "react-router-dom"
import { useFetch } from "../hooks/useFetch"

export default function Article() {
  const { id } = useParams()
  const url = 'http://localhost:3000/articles/' + id
  const { data: article, isPending, error } = useFetch(url)
  const history = useHistory()

  useEffect(() => {
    if (error) {
      // redirect
      // history.goBack()
      setTimeout(() => {
        history.push('/')
      }, 2000)
    }
  }, [error])
```

The useHistory hook allows you access the history instance used by React Router. Using the history instance you can redirect users to another page. The history instance created by React Router uses a Stack(called “History Stack”), that stores all the entries the user has visited.

hystory.push() is used to navigatr to a particular page

Redirect Component

Whenever user enters a URL that doesnot have a route, we will display an emply page


```
import { BrowserRouter, Route, Switch, NavLink, Redirect } from 'react-router-dom'

<Switch>
  <Route exact path="/">
    <Home />
  </Route>
  <Route path="/about">
    <About />
  </Route>
  <Route path="/contact">
    <Contact />
  </Route>
  <Route path="/articles/:id">
    <Article />
  </Route>
  <Route path="*">
    <Redirect to="/" />
  </Route>
</Switch>
```

What we can do is diaplay 404 error or Redirect to some other component using the builtin Redirect component

path="*" means accept any path
We are placing this route at last because this is evaluated only when none of the routes above have a match coz react evaluated routes from top to bottom one abter the other

Query Parameters

```jsx
import { useLocation } from "react-router-dom"

export default function Contact() {
  const queryString = useLocation().search
  console.log(queryString)

  const queryParams = new URLSearchParams(queryString)
  const name = queryParams.get("name")

  //?name=mario

  return (
    <div>
      <h2>Hey {name}, Contact us here...</h2>
      <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Ad nulla tempore est
      expedita nisi eveniet odio dicta. Tempora expedita sint dicta, ducimus
      necessitatibus, at magni aut porro, quis nam explicabo!</p>
    </div>
  )
}
```

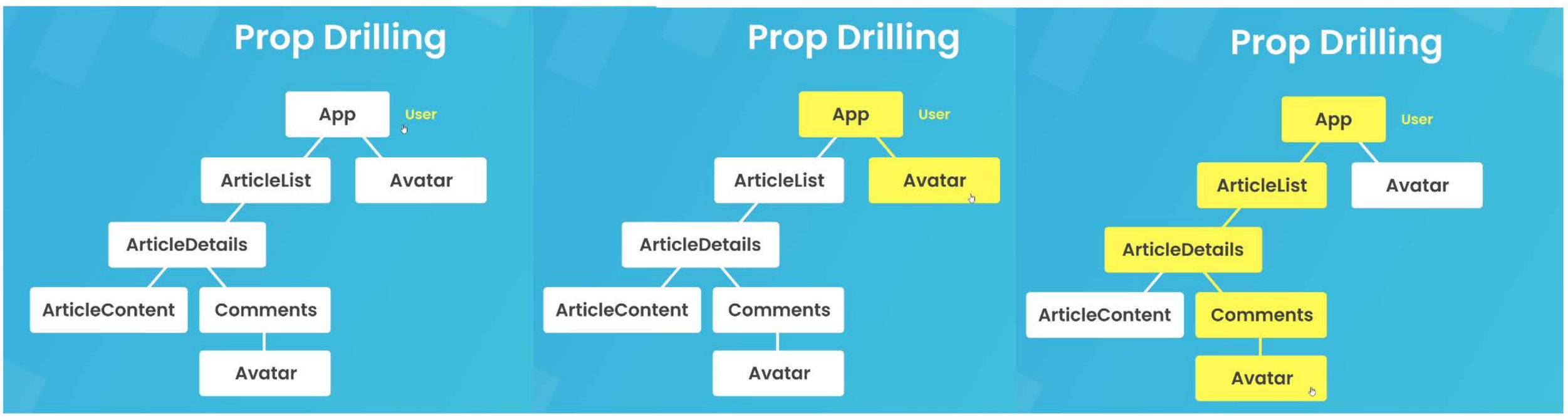
returns a string which contains the queries of the current url

Eg.
localhost:3000/about?name=mario
queryString = name=mario

Used to create key->Value pair of a query
namr->mario

This hook returns the location object used by the react-router. This object represents the current URL and is immutable. Whenever the URL changes, the useLocation() hook returns a newly updated location object. Some of its use includes extracting the query parameters from the URL and doing something depending on the query parameters. The “search” property of the location object returns a string containing the query part of the URL.

Prop Drilling



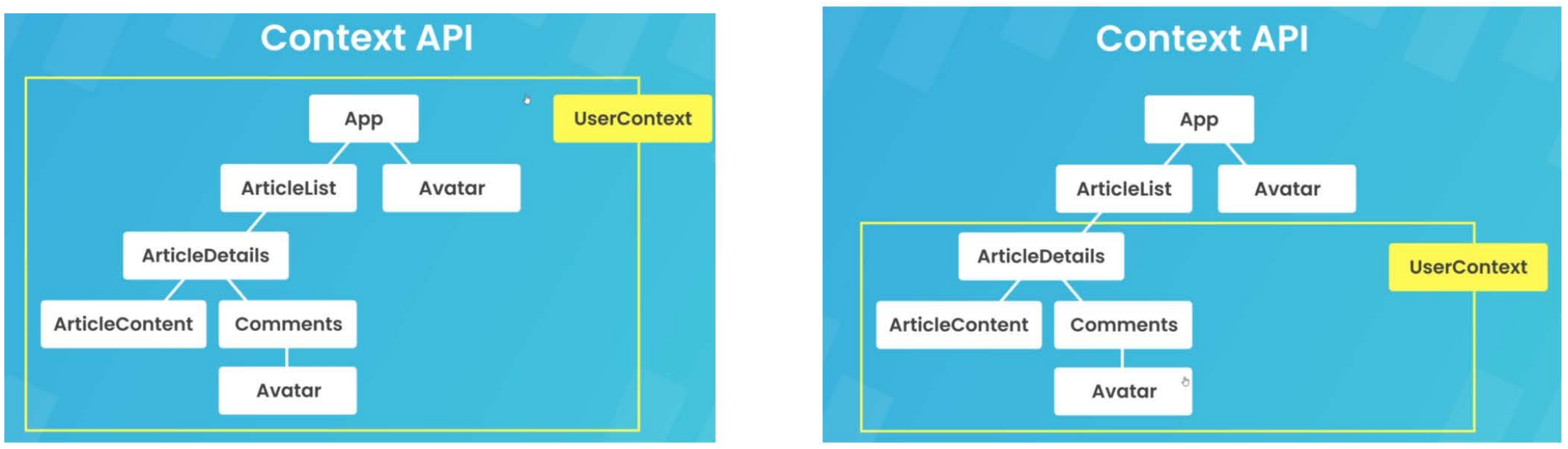
Prop drilling in React refers to the process of passing data down through multiple levels of nested components via props, even if some of the intermediate components do not actually need that data themselves. This can happen when a component needs to access data or a function that is several levels up in the component tree.

Prop drilling can make the code harder to read and maintain, especially when there are many layers of components involved. It can also lead to performance issues, as props have to be passed down through each intermediate component, even if they are not used.

To avoid prop drilling, React provides several alternatives, such as context, which allows data to be shared between components without the need to pass it through each intermediate component.

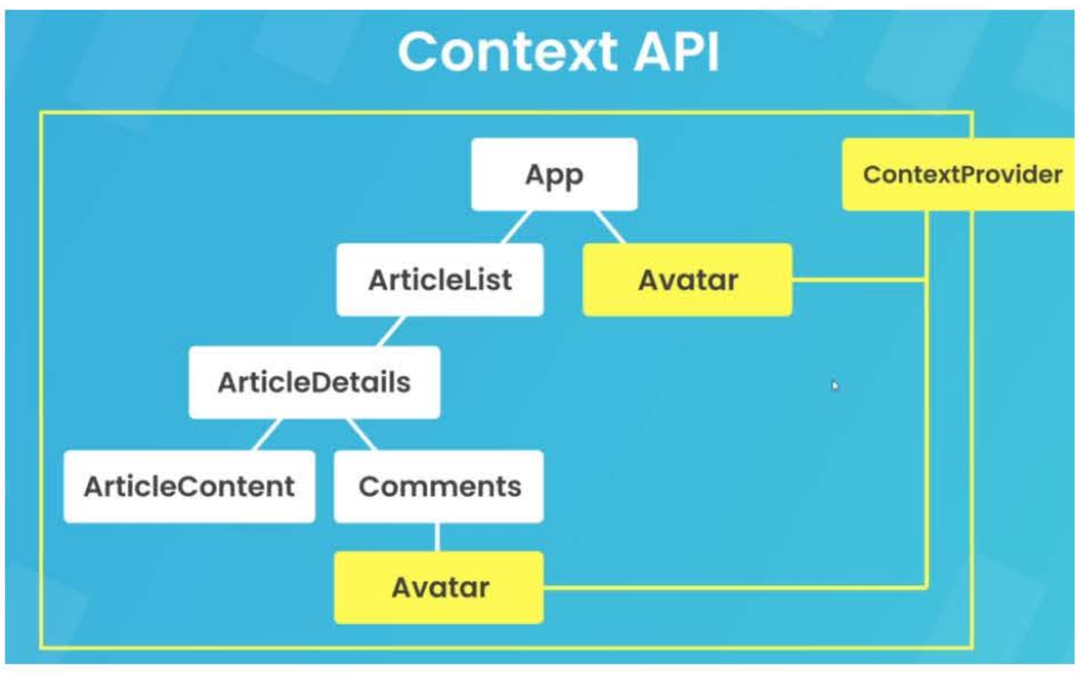
Context API

Context API is a feature in React that allows data to be shared between components without having to pass props down through each intermediate component. It provides a way to pass data through the component tree without having to manually pass props at every level.



So what context does for us is it gives us a central place of storage of stored data and that context can be applied to any component tree of your application.
It can surround the whole application and then any component in that application can access that OR it can just wrap a smalller tree of the application so that it can be shared only in that tree.

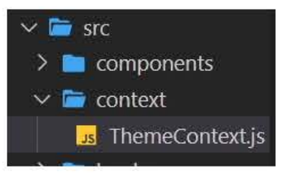
Now when we wrap the context aroud the tree of components , we dont actually wrap the context itself around it. Instead we wrap whats known as **context provider** around it.
Context consists of two main components: the context object, which holds the data, and the context provider, which makes the data available to the components that need it.



When it comes to the context API. Just because you can use global state, does it mean you always should.Because using a lot of global state like this can lead to unnecessary component or renders, because **any component that consumes a context will re-render any time any data in that context changes.**
So don't always reach for it right away. **A bit of prop drilling isn't always a bad thing.**
Use context only when you need to share some kind of global state in a number of different components that are nested at different levels in your component tree.

Creating a Context and Provider

The first step is to create a file for context inside the src folder



The **createContext** function creates a new context for us which can be used to wrap a component tree

src > context > ThemeContext.js > ThemeContext

```
1  import { createContext } from 'react'
2
3  export const ThemeContext = createContext()
```

src > index.js > ...

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import { ThemeContext } from './context/ThemeContext'
6
7  ReactDOM.render(
8    <React.StrictMode>
9      <ThemeContext.Provider value={{ color: 'blue' }}>
10       <App />
11     </ThemeContext.Provider>
12   </React.StrictMode>,
13   document.getElementById('root')
14 );
15
```

index.js

Provides a global context value to the entire application because its wraping the root component ie **<App>**

Although this way of wrapping up the component is completely fine. You'll find most react developers do a slightly different way. And the reason they do it a different way is to make it easier to update contact values in the future, because it may change at some point, right?

So a better way to do this is to create a function called theme provider, which will just be a react component.

src > context > ThemeContext.js > ThemeProvider

ThemeContext.js

```
1  import { createContext } from 'react'
2
3  export const ThemeContext = createContext()
4
5  export function ThemeProvider({ children }) {
6
7    return (
8      <ThemeContext.Provider value={{ color: 'blue' }}>
9        {children}
10     </ThemeContext.Provider>
11   )
12 }
```

The children prop just represents any children components that this component might wrap in the future . It means that all the children components get access to the theme context providing value

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { ThemeProvider } from './context/ThemeContext'

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

In this case, the app component will be the value of the children prop.

We're just doing the same thing as before when we directly wrapped our provider around the app component inside the index file.
Only this time we've bundled that functionality into a custom theme provide a component, but the end result is the same. The provider still just wraps the app component.

So why if we do it this way, which seems like more work to setup?
Well, now it means that we can put some custom logic inside these components, and that custom logic can keep track of the values that we put into the provider. And it can change the value if we need to as well. So this way of working gives us much more flexibility with our contact value in the future. And we're going to see all that in action later on.

Accessing Context Values

```jsx
import { useContext } from 'react'
import { ThemeContext } from '../context/ThemeContext'
```

```jsx
const { color } = useContext(ThemeContext)
```

Accesssing context value in a component which is inside the boundary of context provider

useContext hook gives us accesss to use the context value in any of the child component

```jsx
import { createContext } from 'react'

export const ThemeContext = createContext()

export function ThemeProvider({ children }) {

  // custom logic

  return (
    <ThemeContext.Provider value={{ color: 'blue' }}>
      {children}
    </ThemeContext.Provider>
  )
}
```

ThemeContext.js

Creating a Custom Context Hook

So we've created a context now, and we're consuming the context in the nav bar component and the way we consume in that context is absolutely fine.
We can continue to do that if we want to. However, what we can do is wrap the complex consumer in a custom hook. But why?
First, it's going to reduce the amount of code that we need to write in our components when we use the context.
And second, it's going to allow us to add extra functionality inside the hook as well.

```jsx
import { useContext } from 'react'
import { ThemeContext } from '../context/ThemeContext'

export const useTheme = () => {
  const context = useContext(ThemeContext)

  if (context === undefined) {
    throw new Error("useTheme() must be used inside a ThemeProvider")
  }

  return context
}
```

useTheme.js

Context is undefined when we try to access it outside the scope it

```jsx
import { useTheme } from '../hooks/useTheme'

const { color } = useTheme()
```

Now this is how we can accesss the context value inside the component

Recuders and useReducer hook

So we have created a context, and the context provider is putting a value into our components. We can access that value in any component wrapped by this provider by using either the useContact hook or the custom hook that we created.

But what if at some point we want to be able to change this context value, for example, changing the color from blue to pink or something like that?

Well, what we can do is we can wrap the object value in some kind of states (useState) and then pass the state object as a value in the provider. Then when that state value is changed in the future, the value we pass into the provider is updated too and therefore any component which consumes the context gets that updated value, which causes a re-render of that component with that new value.

But a lot of the time when you're working with context and complex state and different ways of updating states, it is better to use something called a **reducer** instead.

The **useReducer** hook in React is a powerful tool for managing complex state in your components, and it can often be a better choice than using the useState hook in certain situations.

Here are some reasons why you might choose to use useReducer instead of useState:

1. Complex state management: If you have complex state logic that involves multiple values that are interdependent, it can be difficult to manage with useState. In contrast, useReducer allows you to write a reducer function that handles all state updates in a centralized way, making it easier to reason about and maintain.

2. Predictable updates: When using useState, it's possible to accidentally overwrite or mutate the current state, which can lead to unexpected behavior or bugs. In contrast, useReducer enforces a strict update cycle where you can only update the state by dispatching an action object to the **reducer function**. This makes state updates more predictable and easier to reason about.

3. Performance optimization: When using useState, every update to the state causes a re-render of the component. In contrast, with useReducer, you have more control over when and how updates happen, which can lead to better performance in some cases.

4. Complex component hierarchy: If you have a complex component hierarchy with many levels of nesting, it can be tedious to pass state down through props at every level. In contrast, useReducer allows you to share state between components without having to pass it down manually.

src > context > ThemeContext.js > ThemeProvider > changeColor > type

```
1  import { createContext, useReducer } from 'react'
2
3  export const ThemeContext = createContext()
4
5  const themeReducer = (state, action) => {
6    switch (action.type) {
7      case 'CHANGE_COLOR':
8        return { ...state, color: action.payload }
9      default:
10       return state
11   }
12 }
13
14 export function ThemeProvider({ children }) {
15   const [state, dispatch] = useReducer(themeReducer, {
16     color: 'blue'
17   })
18
19   const changeColor = (color) => {
20     dispatch({ type: 'CHANGE_COLOR', payload: color })
21   }
22
23   return (
24     <ThemeContext.Provider value={{...state, changeColor}}>
25       {children}
26     </ThemeContext.Provider>
27   )
28 }
```

new state

action object

We are declaring the reducer function outside the component because it doesnt need to be in that component and get recreated every time the component is evaluated

Reducer function → will be used to update the state

Initial state

The dispatch function is a method that you can call to update the state by sending an **action object** to the reducer function
action object contains two arguements ie
type -> type of state change
payload -> data whose state is to be changed

Before we're just passing this object in that never changes. We want to pass in the state object, which can change and also the change color function so that we can call that function from all the components.

When we use this dispatch function, React looks at the reducer function associated with that dispatch and then it fires that function to make the state change inside it.

Reducer function when it's called using the dispatch function, takes in two arguments the current up to date state and the action object which we passed into the dispatch call.

So then we can use both of those two things to update the state.

So now everythig is set up and if we want to change the value of color, we can call the **changeColor** fuction by passing the new value

```
const { color, changeColor } = useTheme()

changeColor('pink')
```

This is how we can access the context value and change its value if required in any of the component

So whenever this value changes, all the components that use this value will be re-rendered just like the useState hook

Light and Dark Theme

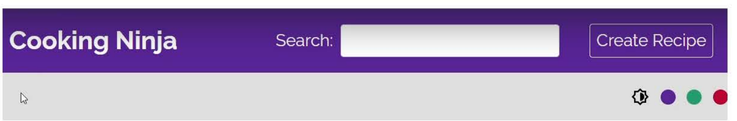
```jsx
import { createContext, useReducer } from 'react'

export const ThemeContext = createContext()

const themeReducer = (state, action) => {
  switch (action.type) {
    case 'CHANGE_COLOR':
      return { ...state, color: action.payload }
    case 'CHANGE_MODE':
      return { ...state, mode: action.payload }
    default:
      return state
  }
}

export function ThemeProvider({ children }) {
  const [state, dispatch] = useReducer(themeReducer, {
    color: '#58249c',
    mode: 'dark'
  })

  const changeColor = (color) => {
    dispatch({ type: 'CHANGE_COLOR', payload: color })
  }
  const changeMode = (mode) => {
    dispatch({ type: 'CHANGE_MODE', payload: mode })
  }

  return (
    <ThemeContext.Provider value={{...state, changeColor, changeMode }}>
      {children}
    </ThemeContext.Provider>
  )
```

ThemeContext.js

```jsx
mponents > ThemeSelector.js > ThemeSelector > toggleMode
import { useTheme } from "../hooks/useTheme"
import modeIcon from '../assets/mode-icon.svg'

// styles
import './ThemeSelector.css'

const themeColors = ['#58249c', '#249c6b', '#b70233']

export default function ThemeSelector() {
  const { changeColor, changeMode, mode } = useTheme()

  const toggleMode = () => {
    changeMode(mode === 'dark' ? 'light' : 'dark')
  }

  return (
    <div className="theme-selector">
      <div className="mode-toggle">
        <img
          onClick={toggleMode}
          src={modeIcon}
          alt="dark/light toggle icon"
          style={{ filter: mode === 'dark' ? 'invert(100%)' : 'invert(20%)' }}
        />
      </div>
      <div className="theme-buttons">
        {themeColors.map(color => (
          <div
            key={color}
            onClick={() => changeColor(color)}
            style={{ background: color }}
          />
        ))}
      </div>
    </div>
```

filter is a property used in css which allows us to use the **invert** fuinction to invert color
invert(100%) : take curr color (black) and invert it to its opp color(white)

ThemeSelector.js



Now we can add styles for dark and light themes

```jsx
import { useTheme } from './hooks/useTheme'

function App() {
  const { mode } = useTheme()

  return (
    <div className={`App ${mode}`}>
      <BrowserRouter>
        <Navbar />
        <ThemeSelector />
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
```

App.js

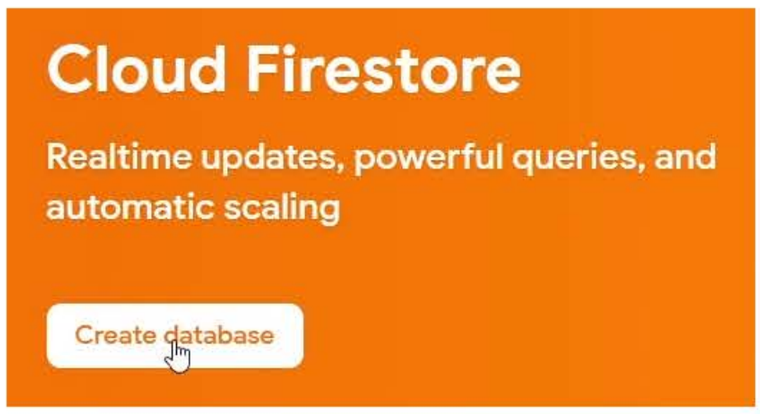
We are adding another class which indicated **mode** in App component

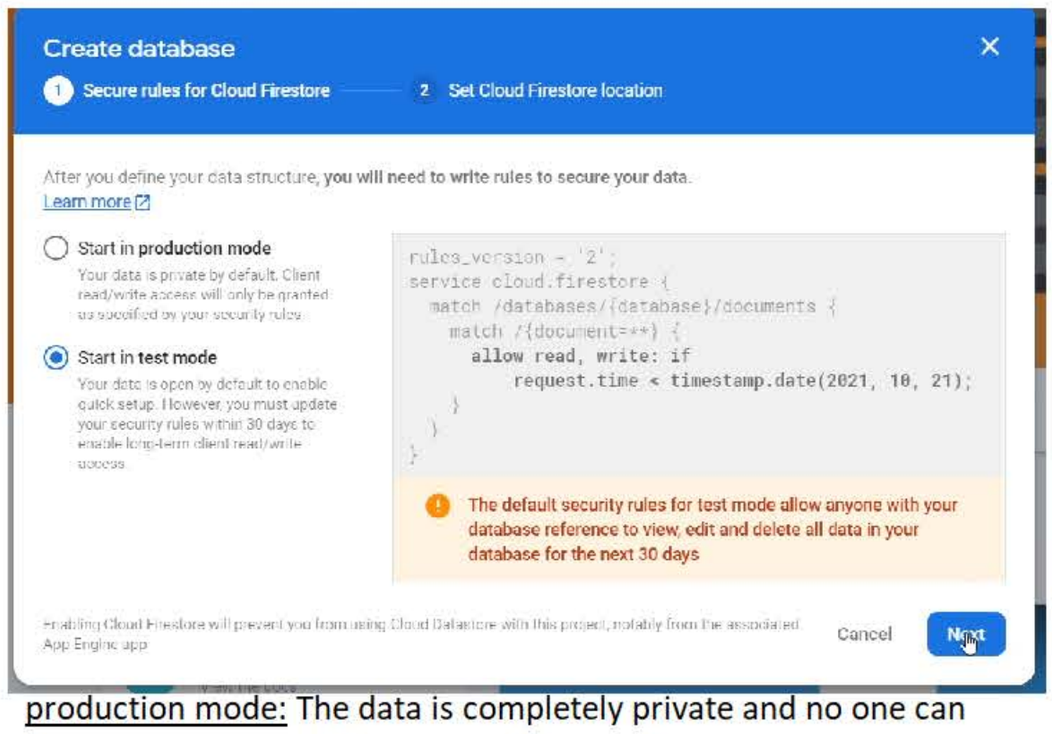
```css
.App {
  min-height: 100%;
}
.App.dark {
  background: #333;
}
```

We can do the same to every component and element which needs to be styled differently in light and dark mode

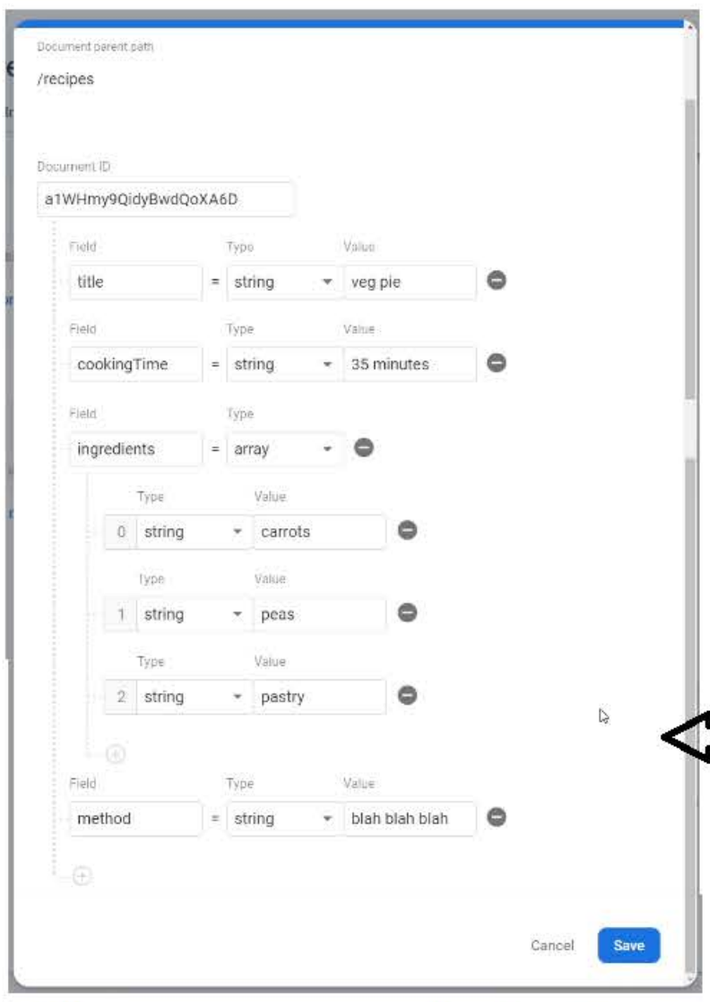
Setting up the database

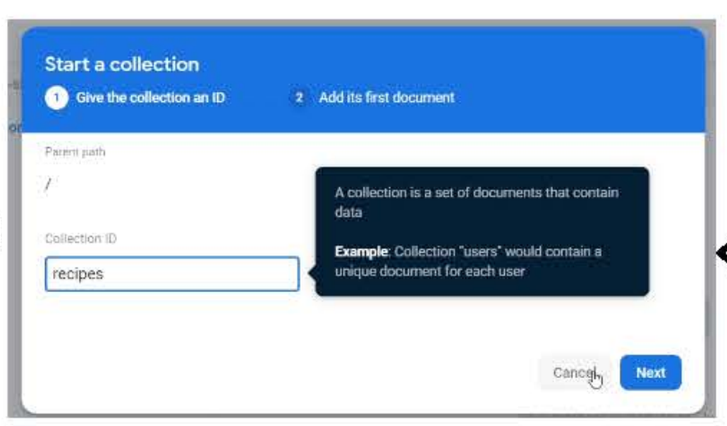
After we create a project, we need to setup the database
There are two types of database (both noSql)
1. Firestore database
2. Realtime database
Realtime database is the original database that firebase created. We will be using the Firestore database which is newer and easier to use.

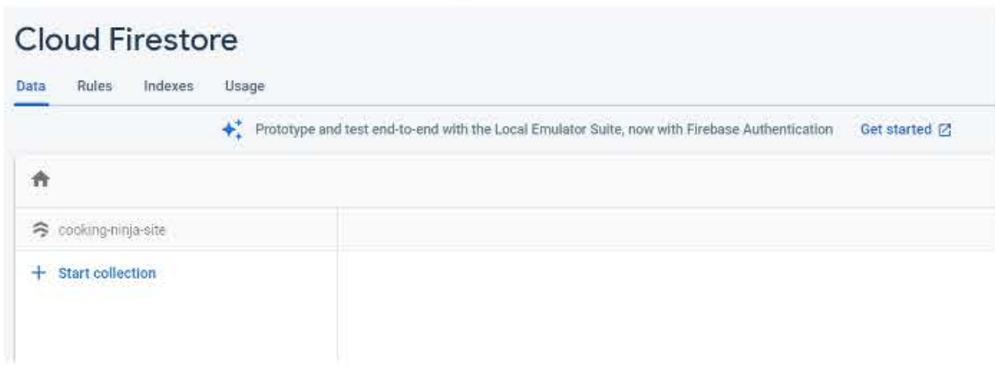




<u>production mode:</u> The data is completely private and no one can have access to our database including out frontend
<u>testmode:</u> Anyone who knows about our database can access the data

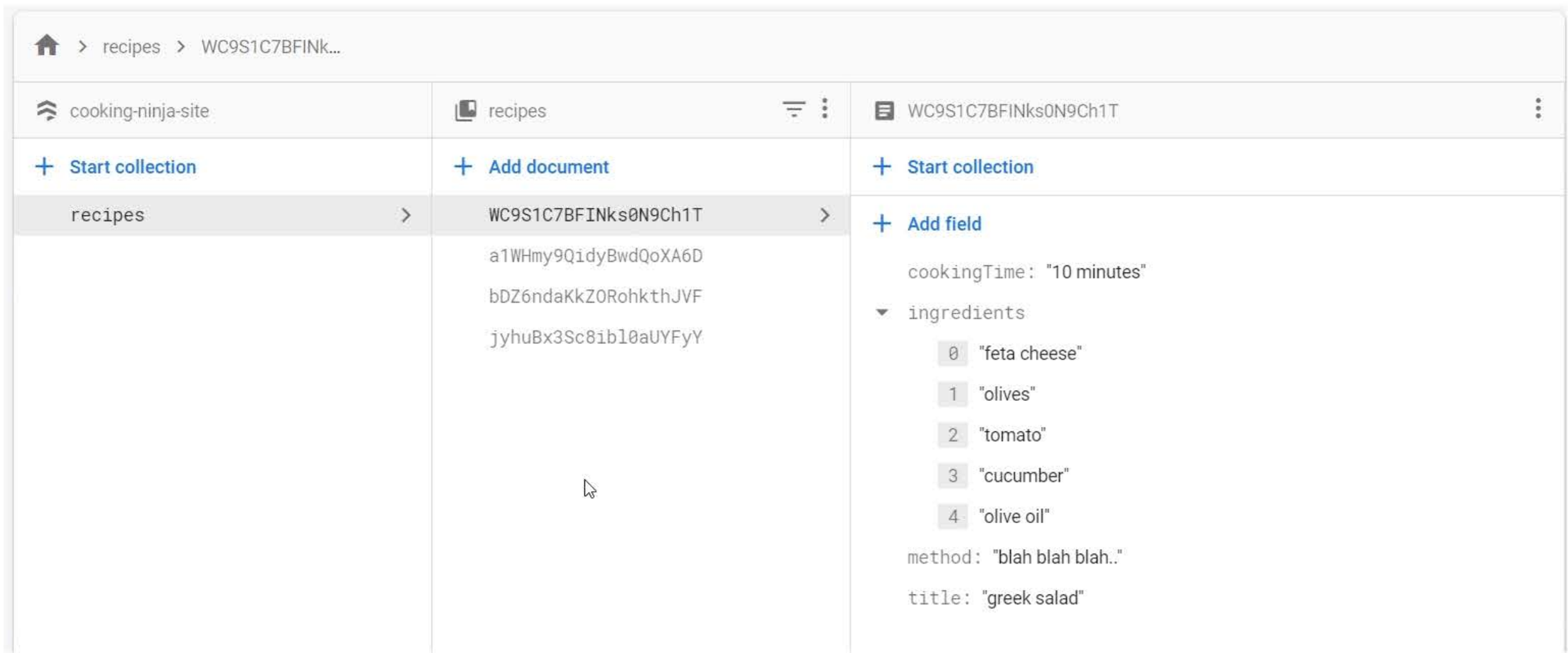






<u>Collections:</u> Equivalent to **table** in sql

When we create a collection firebase asks us to create our first document. Remember each document in a collection must have a unique id which can be used to fetch the document later. A document represents a single record in a collection (eg: single recepie). Document is nothing but a row of a table

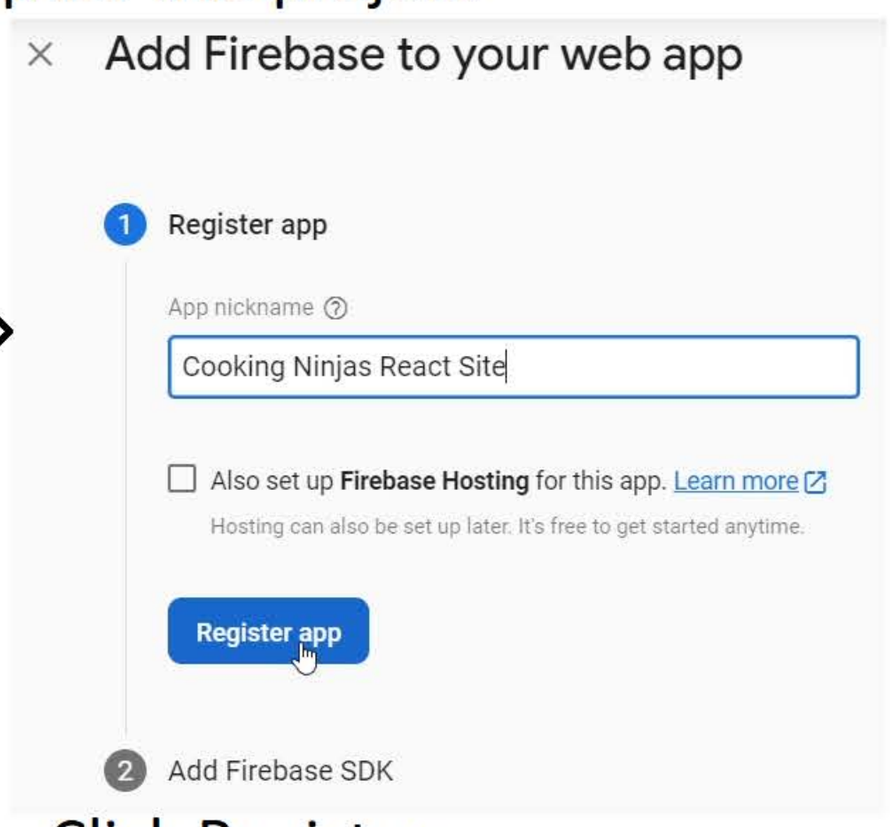


Connecting to Firebase

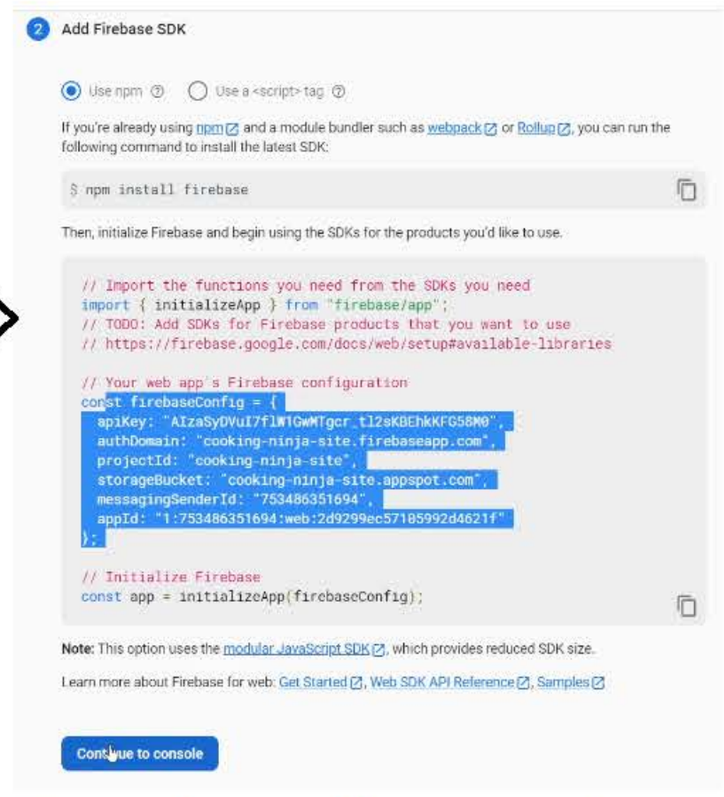
```
PS C:\Users\Shaun\Documents\Tuts\cooking-ninja> npm install firebase@8.5
```

Installing Firebase locally

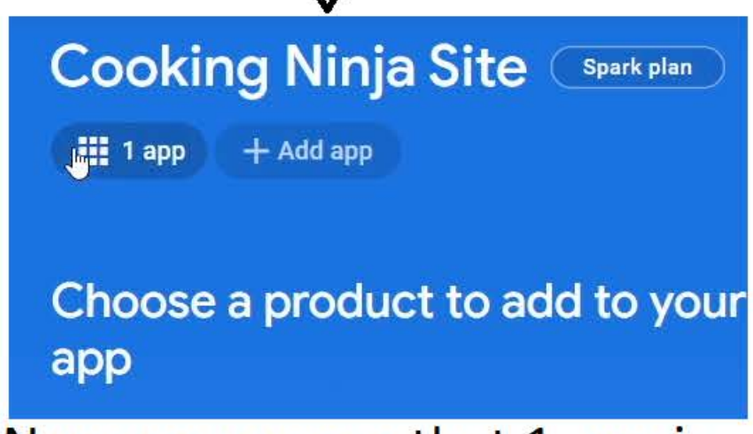
Regestering a Frontend app for this project



Click Register

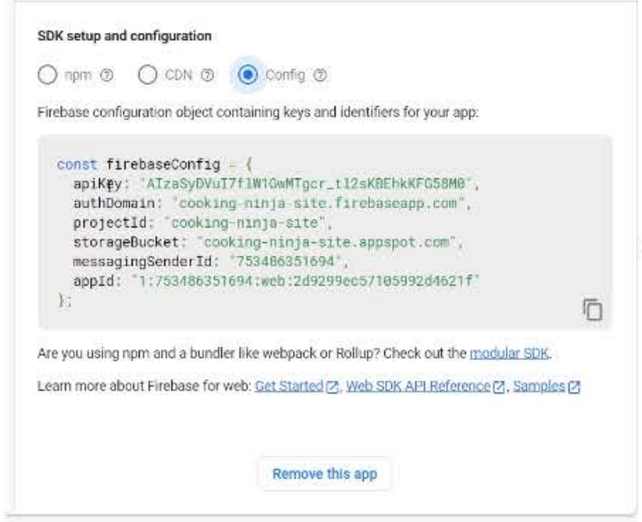


Dont do anything now, just click continue to console



Now we can see that 1 app is regestered for this database

If we click on that and go to settings



We get this page

Copy the config object


```
import firebase from 'firebase/app'
import 'firebase/firestore'

const firebaseConfig = {
  apiKey: "AIzaSyDVuI7flW1GwMTgcr_tl2sKBEhkKFG58M0",
  authDomain: "cooking-ninja-site.firebaseapp.com",
  projectId: "cooking-ninja-site",
  storageBucket: "cooking-ninja-site.appspot.com",
  messagingSenderId: "753486351694",
  appId: "1:753486351694:web:2d9299ec57105992d4621f"
}

// init firebase
firebase.initializeApp(firebaseConfig)

// init services
const projectFirestore = firebase.firestore()

export { projectFirestore }
```

Connects to firebase backend

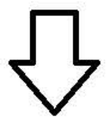
Initializes the firestore service

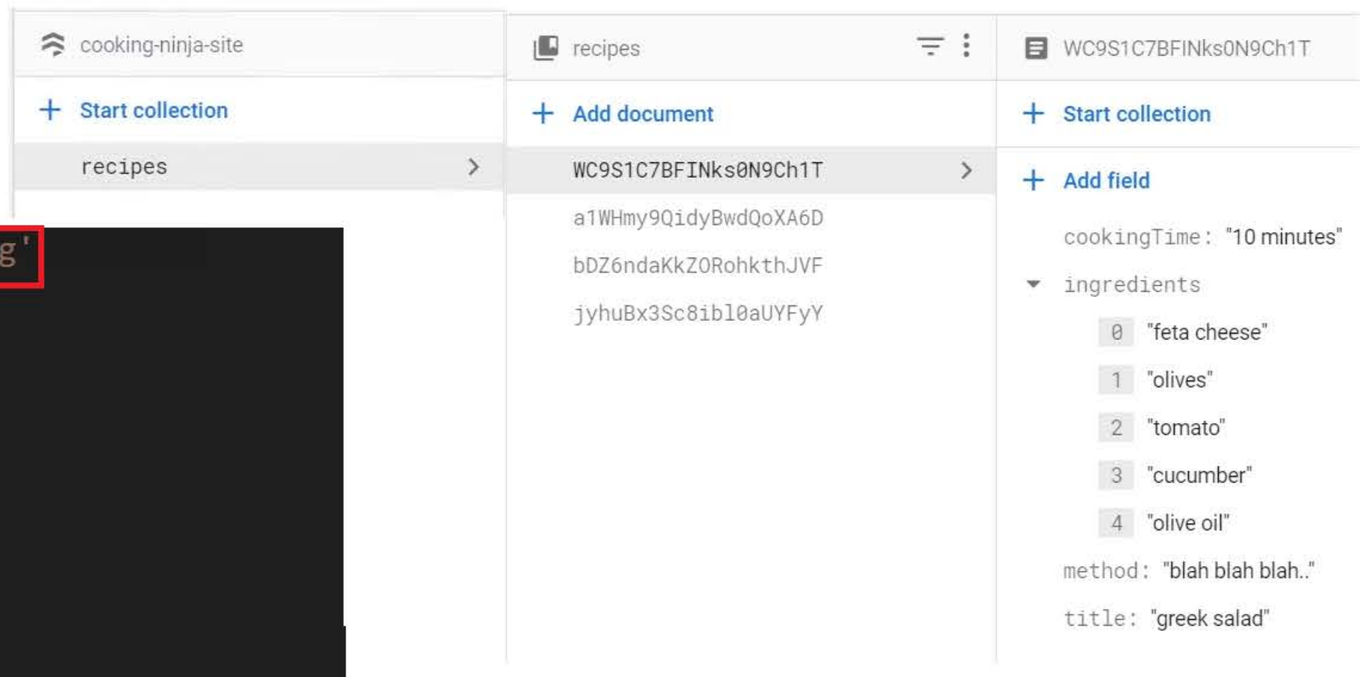
projectFirestore can be used to interact with the database

Fetching a Firestore Collection

```jsx
import { useFetch } from '../../hooks/useFetch'

//styles
import './Home.css'

// components
import RecipeList from '../../components/RecipeList'

export default function Home() {
  const { data, isPending, error } = useFetch('http://localhost:3000/recipes')

  return (
    <div className="home">
      {error && <p className="error">{error}</p>}
      {isPending && <p className="loading">Loading...</p>}
      {data && <RecipeList recipes={data} />}
    </div>
  )
}
```

We used the **useFetch()** hook along with **JSON server** in the Home.js component in our **recepieDirectory** project. Now we will be using the Firestore database now






```jsx
import { projectFirestore } from '../../firebase/config'
import { useEffect } from 'react'

//styles
import './Home.css'

// components
import RecipeList from '../../components/RecipeList'

export default function Home() {
  const [data, setData] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const [error, setError] = useState(false)

  useEffect(() => {
    setIsPending(true)

    projectFirestore.collection('recipes').get().then((snapshot) => {
      if (snapshot.empty) {
        setError('No recipes to load')
        setIsPending(false)
      } else {
        let results = []
        snapshot.docs.forEach(doc => {
          results.push({ id: doc.id, ...doc.data() })
        })
        setData(results)
        setIsPending(false)
      }
    }).catch(err => {
      setError(err.message)
      setIsPending(false)
    })

  }, [])

  return (
    <div className="home">
      {error && <p className="error">{error}</p>}
      {isPending && <p className="loading">Loading...</p>}
      {data && <RecipeList recipes={data} />}
    </div>
  )
}
```

True => Error

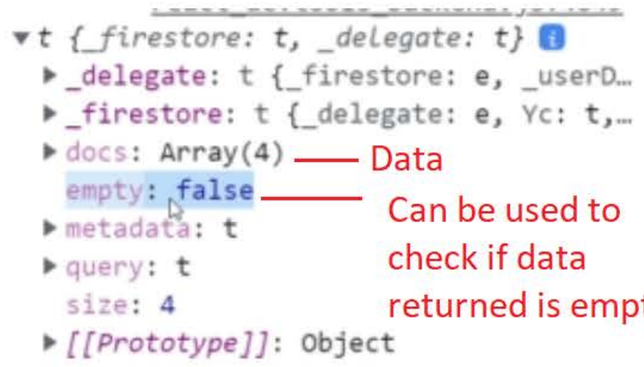
Single Recepie

This is an asynchronous function and hence returns a promise

Used to get the data

Connects to 'recepies' colection

Snapshot of the recipes collection



Data

Can be used to check if data returned is empty

Fetching a Document From A Collection

Everything on the homepage is working fine, but when we click on a particular recepie, the recepie wont load because we are still using the useFetch( ) hook to fetch the data from JSON server in the recipe component.

```
const { id } = useParams()
const url = 'http://localhost:3000/recipes/' + id
const { error, isPending, data: recipe } = useFetch(url)
const { mode } = useTheme()
```

```
const { id } = useParams()
const { mode } = useTheme()

const [data, setData] = useState(null)
const [isPending, setIsPending] = useState(false)
const [error, setError] = useState(false)

useEffect(() => {
  setIsPending(true)

  projectFirestore.collection('recipes').doc(id).get().then((doc) => {
    if (doc.exists) {
      setIsPending(false)
      setRecipe(doc.data())
    } else {
      setIsPending(false)
      setError('Could not find that recipe')
    }
  })

}, [id])
```

Create.js

Uploading New Data to Firestore Database

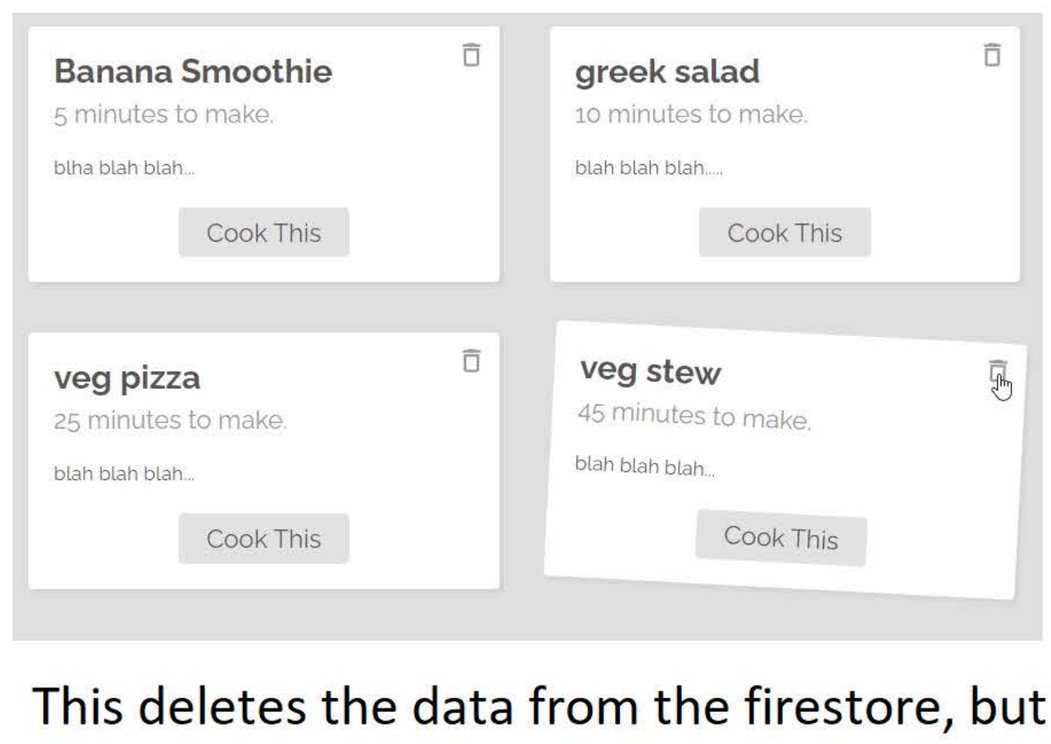
We used the POST method to upload data to the JSON server in Create component using the useFetch( ) hook earlier

```
const [title, setTitle] = useState('')
const [method, setMethod] = useState('')
const [cookingTime, setCookingTime] = useState('')
const [newIngredient, setNewIngredient] = useState('')
const [ingredients, setIngredients] = useState([])
const ingredientInput = useRef(null)
const history = useHistory()

const { postData, data, error } = useFetch('http://localhost:3000/recipes', 'POST')

const handleSubmit = (e) => {
  e.preventDefault()
  postData({ title, ingredients, method, cookingTime: cookingTime + ' minutes' })
}
```

```
const [title, setTitle] = useState('')
const [method, setMethod] = useState('')
const [cookingTime, setCookingTime] = useState('')
const [newIngredient, setNewIngredient] = useState('')
const [ingredients, setIngredients] = useState([])
const ingredientInput = useRef(null)
const history = useHistory()

const handleSubmit = async (e) => {
  e.preventDefault()
  const doc = { title, ingredients, method, cookingTime: cookingTime + ' minutes' }

  try {
    await projectFirestore.collection('recipes').add(doc)
    history.push('/')
  } catch(err) {
    console.log(err)
  }
}
```

Deleting a Document of a Collection

```
const handleClick = (id) => {
  projectFirestore.collection('recipes').doc(id).delete()
}

return (
  <div className="recipe-list">
    {recipes.map(recipe => (
      <div key={recipe.id} className={`card ${mode}`}>
        <h3>{recipe.title}</h3>
        <p>{recipe.cookingTime} to make.</p>
        <div>{recipe.method.substring(0, 100)}...</div>
        <Link to={`/recipes/${recipe.id}`}>Cook This</Link>
        <img
          className="delete"
          src={Trashcan}
          onClick={() => handleClick(recipe.id)}
        />
      </div>
    ))}
  </div>
```



This deletes the data from the firestore, but for the change to be visible in the forntend, we have to refresh the page. To avoid this we need to learn about realime data

Real-Time Collection Data

Now a way to combat the problem above is to use what's known as Real-Time Collection data, and what that does for us is essentially sets up a listener to our collection ie our recipes collection, and it all listens to events that go on inside this collection.

Now, one of those events would be to delete some data, delete a document, and when that happens,
Firestore will automatically send us back another snapshot. And in our application, we can listen for that snapshot. And when we receive that snapshot, we can look at the documents on it and we can update our local state to match that snapshot.

So when we delete something, then it should update in the UI as well, and that would be the same for adding documents as well.

```
useEffect(() => {
  setIsPending(true)

  projectFirestore.collection('recipes').get().then((snapshot) => {
    if (snapshot.empty) {
      setError('No recipes to load')
      setIsPending(false)
    } else {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({ id: doc.id, ...doc.data() })
      })
      setData(results)
      setIsPending(false)
    }
  }).catch(err => {
    setError(err.message)
    setIsPending(false)
  })

}, [])
```

We used the .get method to fetch the recipies on home page

```
useEffect(() => {
  setIsPending(true)

  const unsub = projectFirestore.collection('recipes').onSnapshot((snapshot) => {
    if (snapshot.empty) {
      setError('No recipes to load')
      setIsPending(false)
    } else {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({ id: doc.id, ...doc.data() })
      })
      setData(results)
      setIsPending(false)
    }
  }, (err) => {
    setError(err.message)
    setIsPending(false)
  })

  return () => unsub()

}, [])
```

CLEANUP FUNC

Home.js

The data is fetched initially as usual. But thereafter whenever there is a change in the recipes collection, its gonna fire a snapshot event

The **onSnapshot** method takes in two arguements
1. Function to be fired on any change in collecttion
2. Function to be fired when we have an error

The listener is active evem when we navigate away from the homepage and is still listening for changes. If a change occurs, it will update the state of the results on homepage even though we are on a different page. So we need a cleanup function and it will fire when the homepage component unmounts.
The onSnapshot method returns an unsubscribe function which when invoked unsubscribes us from the snapshot .

The function returned by the useEffect() hook is nothing but the cleanup function and will be invoked automatically invoked when the component unmounts.

Updating The Database

```
projectFirestore.collection('recipes').doc(id).update({
  title: 'Something completely different'
})
```

This is just a vague way of updating the database. To do it much better, we can use a form.
As seen before, to see the change on the frontend, we have to refresh the page

Realtime Document Data

While fetching an individual recipe, we used the **.get** method. Instead, we have to use the **onSnapshot() method**

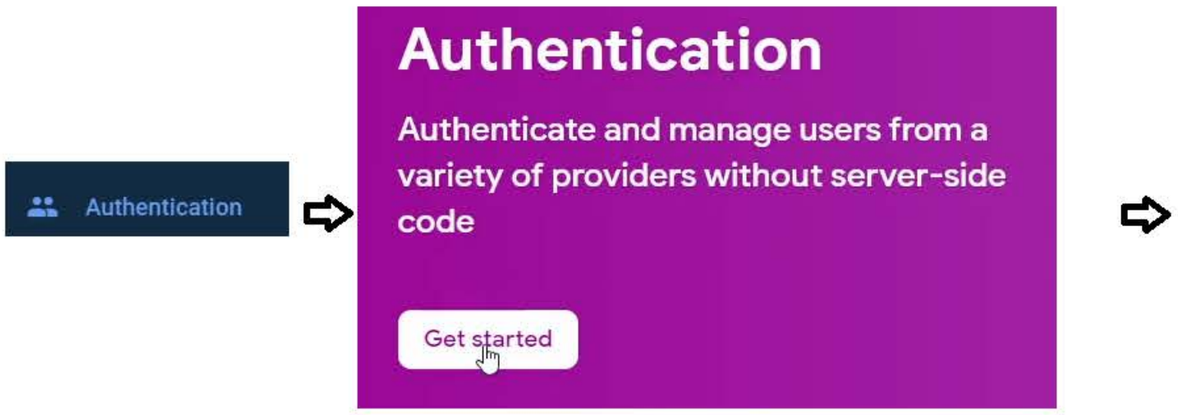
```
useEffect(() => {
  setIsPending(true)

  const unsub = projectFirestore.collection('recipes').doc(id).onSnapshot((doc) => {
    if (doc.exists) {
      setIsPending(false)
      setRecipe(doc.data())
    } else {
      setIsPending(false)
      setError('Could not find that recipe')
    }
  })

  return () => unsub()

}, [id])
```

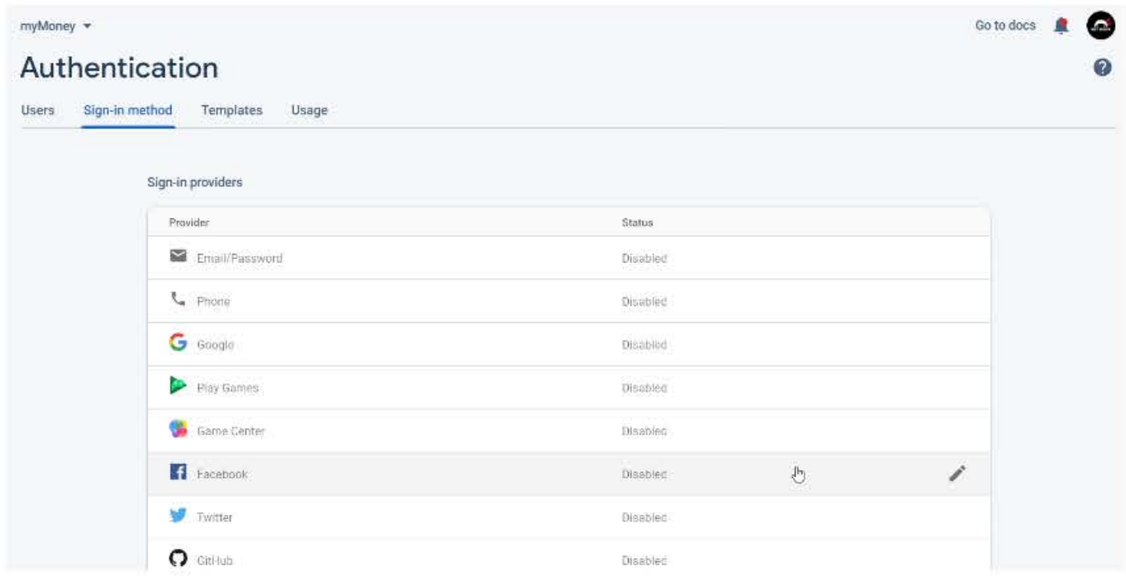
Create.js

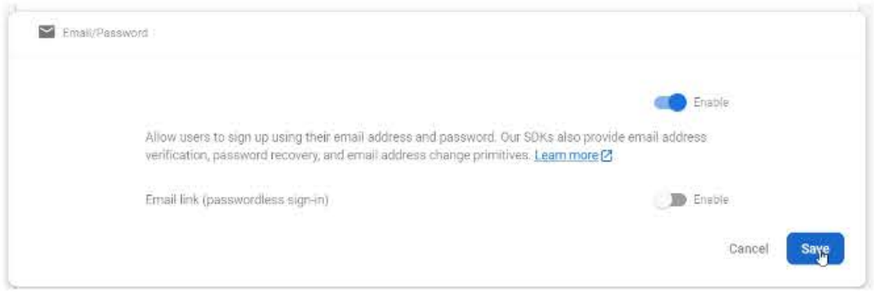
Creating a hook

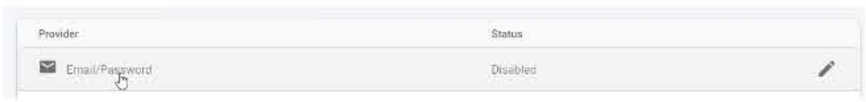
```
import { useEffect, useState } from "react"      You, 44 minutes ago • Readme Modified
import { projectForestore } from "../firebase/config"

export const useFetchFirestore = (collection, id, searchParam) => {
    const [data, setData] = useState(null)
    const [isPending, setIsPending] = useState(false)
    const [error, setError] = useState(false)

    useEffect(() => {
        setIsPending(true)
        let unsub;

        if(id != null){ //Fetching a particular recipe
            unsub = projectForestore.collection(collection).doc(id).onSnapshot((doc) => {
                if(doc.exists){
                    setIsPending(false)
                    setData(doc.data())
                }
                else{
                    setIsPending(false)
                    setError('Coudnot find the recipe')
                }
            })
        }
        else {
             unsub = projectForestore.collection(collection).onSnapshot((snapshot) => {
                if(snapshot.empty){
                    setError('No recipes to load')
                    setIsPending(false)
                }
                else{
                    let results = []
                    snapshot.docs.forEach(doc => {
                        if(searchParam != null ){ //Searched recipes
                            if(doc.data().title.toLowerCase().includes(searchParam.toLowerCase())){
                                results.push({id: doc.id, ...doc.data()})
                            }
                        }
                        else{//Fetching all recipes
                            results.push({ id: doc.id, ...doc.data() })
                        }
                    })
                    setData(results)
                    setIsPending(false)
                }
            }, (err) => {
                setError(err.message)
                setIsPending(false)
            })
        }


        return () => unsub()
    }, [id])

    return { data, isPending, error }
}
```

Firebase Authentication Setup










```
import firebase from 'firebase/app'
import 'firebase/firestore'
import 'firebase/auth'

const firebaseConfig = {
  apiKey: "AIzaSyAARDxNZTL216xkei9QgMOiSQ410EW-kmw",
  authDomain: "mymoney-c0596.firebaseapp.com",
  projectId: "mymoney-c0596",
  storageBucket: "mymoney-c0596.appspot.com",
  messagingSenderId: "443730170101",
  appId: "1:443730170101:web:0bcbbf2e4a22a5c6fbf5bc"
}

// init firebase
firebase.initializApp(firebaseConfig)

// init services
const projectFirestore = firebase.firestore()
const projectAuth = firebase.auth()

export { projectFirestore, projectAuth }
```

useSignup( ) hook

This is a custom hook used to sign up a user to the webapp on submitting the signup form.

Firebase authentication allows us to set a user's display name on a user.
We cant just add any properties which we want. There are certain properties that Firebase authentication allows us to store and **displayName** is one of them.
Another one is a photo URL for some kind of thumbnail, and we're going to see that in the final project, but you can't just go adding any properties to them if you want to add extra properties to a user, we create a Firestore document for that user, and we might see how that works later on.

```
import { useState } from 'react'
import { projectAuth } from '../firebase/config'

export const useSignup = () => {
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)

  const signup = async (email, password, displayName) => {
    setError(null)
    setIsPending(true)

    try {
      // signup user
      const res = await projectAuth.createUserWithEmailAndPassword(email, password)
      console.log(res.user)

      if (!res) {
        throw new Error('Could not complete signup')
      }

      // add display name to user
      await res.user.updateProfile({ displayName })

      setIsPending(false)
      setError(null)

    }
    catch (err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
    }
  }

  return { error, isPending, signup }
}
```

Interacts with firebase authentication

We are setting error to null because if there was some error previously, we want to set it back to null

This reaches out to Firebase Auth sign that user up with an email and a password.
response is gonna be null if user is not created

We cant put the displayName while creating the user. We have to do is we have to create the user first of all, then we have to update the profile of that user and set the display name property.

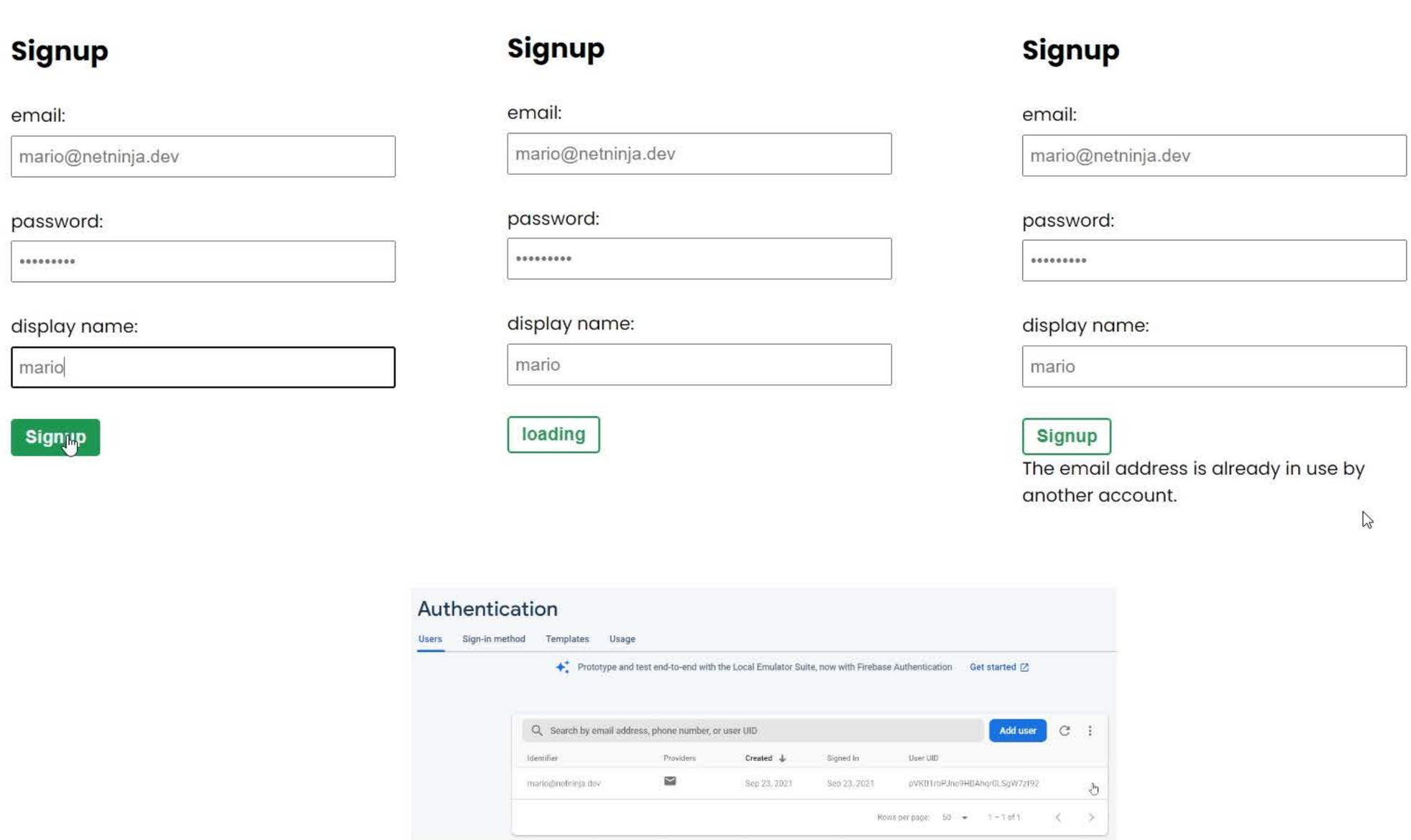
using the useSignup( ) hook

```
const { signup, isPending, error } = useSignup()
signup(email, password, displayName)
```

```
{!isPending && <button className="btn">Signup</button>}
{isPending && <button className="btn" disabled>loading</button>}
{error && <p>{error}</p>}
```

shows only when isPending is false

shows only when isPending is true



Context API for user

After a user signs up, we can see that Firebase returns the user object. Now this huge object contains the token for that user, which Firebase uses to authenticate the user to firebase server. That token is passed back and forth from our website to Firebase on every request for something like data from the database.
We can use that user object in our front end appliction to display content related to only that specific user.
So ideally we want to store thsi user object in some kind of global state when user signs up.So for that we use **context api.**

```
src > context > ThemeContext.js > ThemeProvider > changeColor > type
1  import { createContext, useReducer } from 'react'
2
3  export const ThemeContext = createContext()
4
5  const themeReducer = (state, action) => {
6    switch (action.type) {
7      case 'CHANGE_COLOR':
8        return { ...state, color: action.payload }
9      default:                                          new state
10       return state
11   }
12 }
13
14 export function ThemeProvider({ children }) {
15   const [state, dispatch] = useReducer(themeReducer, {
16     color: 'blue'
17   })
18
19   const changeColor = (color) => {
20     dispatch({ type: 'CHANGE_COLOR', payload: color })
21   }                                                  action object
22
23   return (
24     <ThemeContext.Provider value={{...state, changeColor}}>
25       {children}
26     </ThemeContext.Provider>
27   )
28 }                                                    ThemeContext.js
```

We are declaring the reducer function outside the component because it doesnt need to be in that component and get recreated every time the component is evaluated

Reducer function → will be used to update the state

Initial state

The dispatch function is a method that you can call to update the state by sending an **action object** to the reducer function
action object contains two arguements ie
type -> type of state change
payload -> data whose state is to be changed

Before we're just passing this object in that never changes. We want to pass in the state object, which can change and also the change color function so that we can call that function from all the components.

When we use this dispatch function, React looks at the reducer function associated with that dispatch and then it fires that function to make the state change inside it.

Reducer function when it's called using the dispatch function, takes in two arguments the current up to date state and the action object which we passed into the dispatch call.
So then we can use both of those two things to update the state.

So now everythig is set up and if we want to change the value of color, we can call the **changeColor**

```
import { useContext } from 'react'
import { ThemeContext } from '../context/ThemeContext'

export const useTheme = () => {
  const context = useContext(ThemeContext)

  if (context === undefined) {
    throw new Error("useTheme() must be used inside a ThemeProvider")
  }

  return context
}                                                    useTheme.js
```

Context is undefined when we try to access it outside the scope it

```
const { changeColor, changeMode, mode } = useTheme()
```

using context Api

context api revision

```
import { createContext, useReducer } from 'react'

export const AuthContext = createContext()

export const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { ...state, user: action.payload }
    default:
      return state
  }
}
export const AuthContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, {
    user: null
  })

  return (
    <AuthContext.Provider value={{ ...state, dispatch }}>
      { children }
    </AuthContext.Provider>
  )
}                                                    AuthContext.js
```

```
index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import { AuthContextProvider } from './context/AuthContext';
6
7  ReactDOM.render(
8    <React.StrictMode>
9      <AuthContextProvider>
10       <App />
11     </AuthContextProvider>
12   </React.StrictMode>,
13   document.getElementById('root')
14 );                                                 index.js
```

```
import { AuthContext } from "../context/AuthContext"
import { useContext } from "react"

export const useAuthContext = () => {
  const context = useContext(AuthContext)

  if (!context) {
    throw Error('useAuthContext must be inside an AuthContextProvider')
  }

  return context
}                                                    useAuthContext.js
```

```
import { useState } from 'react'
import { projectAuth } from '../firebase/config'
import { useAuthContext } from './useAuthContext'

export const useSignup = () => {
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const { dispatch } = useAuthContext()

  const signup = async (email, password, displayName) => {
    setError(null)
    setIsPending(true)

    try {
      // signup
      const res = await projectAuth
            .createUserWithEmailAndPassword(email, password)
      if (!res) {
        throw new Error('Could not complete signup')
      }

      // add display name to user
      await res.user.updateProfile({ displayName })

      if (!res) {
        throw new Error('Could not complete signup')
      }

      // add display name to user
      await res.user.updateProfile({ displayName })

      // dispatch login action
      dispatch({ type: 'LOGIN', payload: res.user })

      setIsPending(false)
      setError(null)
    }
    catch(err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
    }
  }

  return { signup, error, isPending }
}                                                    useSignup() hook
```

We are directly using **LOGIN** in dispatch because firebase automatically logs a user in when the user signs up ie creates a new account

Creating a logout hook

```jsx
import { useState } from 'react'
import { projectAuth } from '../firebase/config'
import { useAuthContext } from './useAuthContext'

export const useLogout = () => {
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const { dispatch } = useAuthContext()

  const logout = async () => {
    setError(null)
    setIsPending(true)

    // sign the user out
    try {
      await projectAuth.signOut()

      // dispatch logout action
      dispatch({ type: 'LOGOUT' })

      setIsPending(false)
      setError(null)
    }
    catch(err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
    }
  }

  return { logout, error, isPending }

}
```

useLogout.js

```jsx
export const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { ...state, user: action.payload }
    case 'LOGOUT':
      return { ...state, user: null }
    default:
      return state
  }
}
```

Updating the reducer function in AuthContext.js

using the useLogout( ) hook

```jsx
import { useLogout } from '../hooks/useLogout'

const { logout } = useLogout()

<button className="btn" onClick={logout}>Logout</button>
```

Adding a cleanup function

The reason why we add a cleanup function is the same as discussed before. But previously we used an **AbortController** to do so. But when we are doing something like a login or logout we cannot use that. So we need to do bit more of a manual approach.

WKT the arror is caused when we try to update a state of a component which is already unmounted. So out main job now is to prevent state in that component to be updated is component unmounts.

```
import { useState } from 'react'
import { projectAuth } from '../firebase/config'
import { useAuthContext } from './useAuthContext'

export const useLogout = () => {
  const [isCancelled, setIsCancelled] = useState(false)
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const { dispatch } = useAuthContext()

  const logout = async () => {
    setError(null)
    setIsPending(true)

    // sign the user out
    try {
      await projectAuth.signOut()

      // dispatch logout action
      dispatch({ type: 'LOGOUT' })

      // update state
      if (!isCancelled) {
        setIsPending(false)
        setError(null)
      }
    }
    catch(err) {
      if (!isCancelled) {
        console.log(err.message)
        setError(err.message)
        setIsPending(false)
      }
    }
  }

  useEffect(() => {
    return () => setIsCancelled(true)
  }, [])

  return { logout, error, isPending }
}
```

useLogout.js

We introduce a new state which indicates if the component is unmounted or not

So when we try to logout and do something else in the meantime. The cleanup function is called which will set the **isCancelled** state to be true.

The state of setError and setIsPending will not be updated now

HENCE WE NOT ALLOWING ANY STATE TO BE UPDATED AFTER THE COMPONENT IS UNMOUNTED

CLEANUP FUNC

We are using useEffect becuse cleanup functions can be created only inside the useEffect hook

We can impliment the cleanup function in useSignup and useLogin hook the same way

useLogin hook

```
import { useState, useEffect } from 'react'
import { projectAuth } from '../firebase/config'
import { useAuthContext } from './useAuthContext'

export const useLogin = () => {
  const [isCancelled, setIsCancelled] = useState(false)
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)
  const { dispatch } = useAuthContext()

  const login = async (email, password) => {
    setError(null)
    setIsPending(true)

    // sign the user out
    try {
      const res = await projectAuth.signInWithEmailAndPassword(email, password)

      // dispatch logout action
      dispatch({ type: 'LOGIN', payload: res.user })

      // update state
      if (!isCancelled) {
        setIsPending(false)
        setError(null)
      }
    }
    catch(err) {
      if (!isCancelled) {
        console.log(err.message)
        setError(err.message)
        setIsPending(false)
      }
    }
  }

  useEffect(() => {
    return () => setIsCancelled(true)
  }, [])

  return { login, error, isPending }
}
```

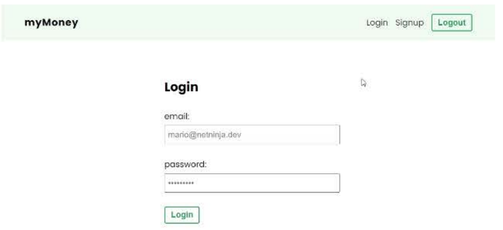
using useLogin hook

```
const { login, error, isPending } = useLogin()
```

```
login(email, password)
```

```
{!isPending && <button className="btn">Login</button>}
{isPending && <button className="btn" disabled>loading</button>}
{error && <p>{error}</p>}
```

Conditionally showing user info

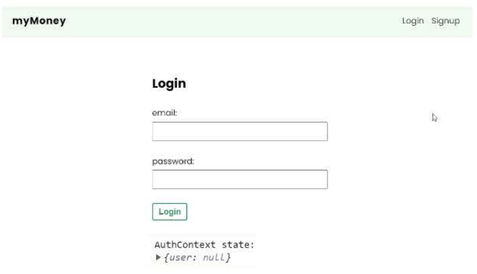


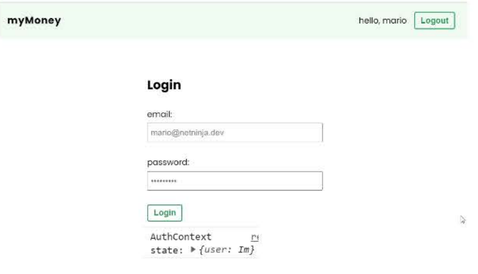
This is what we have done until now
But we want to show login and signup buttons when a user has not logged in and show logout button only after a user logs in.
To do so we can use the user contextAPI which is globally available. user object is null when no user is signed in.

```jsx
import { Link } from "react-router-dom"
import { useLogout } from '../hooks/useLogout'
import { useAuthContext } from "../hooks/useAuthContext"

// styles
import styles from './Navbar.module.css'

export default function Navbar() {
  const { logout } = useLogout()
  const { user } = useAuthContext()

  return (
    <nav className={styles.navbar}>
      <ul>
        <li className={styles.title}>myMoney</li>

        {!user && (
          <>
            <li><Link to="/login">Login</Link></li>
            <li><Link to="/signup">Signup</Link></li>
          </>
        )}

        {user && (
          <>
            <li>hello, {user.displayName}</li>
            <li>
              <button className="btn" onClick={logout}>Logout</button>
            </li>
          </>
        )}
      </ul>
    </nav>
  )
```

We are using fragement because only single root element is allowed in react.

Navbar.js





We have a small problem with this. When we refresh the page even when a user is logged, the page is shown as if the user is logged out and the user object shows null even though a user just logged in. This is because when we refresh the react page, we are recreating all the react components and we set an initial value of authentication context (user obj) is set to null again.

But according to Firebase the user is still logged in. If the user sends a request to the firebase server, it will see the user as logged in, it is just that the auth context is not matching the user who has logged in

Solution:
When our React application first loads, we do not render any component until we've checked with Firebase. If a user is currently logged in, do we have that JSON web token that's valid? And then when that check is complete, Firebase is either going to say, yep, that a user logged in, in which case we can update the auth context state to match that user. Or they're going to say no, a user is not logging in which case will leave the user to be null.

FIrebase Authentication State Change

To perform what we discussed before, we are gonna work inside the AuthContext file

```jsx
import { createContext, useReducer } from 'react'

export const AuthContext = createContext()

export const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { ...state, user: action.payload }
    case 'LOGOUT':
      return { ...state, user: null }
    case 'AUTH_IS_READY':
      return { ...state, user: action.payload, authIsReady: true }
    default:
      return state
  }
}

export const AuthContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, {
    user: null,
    authIsReady: false
  })

  useEffect(() => {
    const unsub = projectAuth.onAuthStateChanged((user) => {
      dispatch({ type: 'AUTH_IS_READY', payload: user })
      unsub()
    })
  }, [])

  return (
    <AuthContext.Provider value={{ ...state, dispatch }}>
      { children }
    </AuthContext.Provider>
  )
}
```

AuthContext.js

Here we want to check if user has logged in or not whenever the react page loads initially. This is why we use the **useEffect** hook with an empty dependency array.

The function is gonna fire every time there is some change in authentication state. But we want the function to fire only once when the page is loaded to find out the initial user. To do this, we use the unsubscribe funtion returned by the onAuthStateChanged function which will unsubscribe from the listener

We are gonna use this inside the App.js file. We are gonna show the component tree only when this is true

This method communicates with firebase and fires the callback function whenthere is some change in authentication status. It will tell if a user has loggedin or not.

If user is logged in, user is set else user is set to null

Above component in action:

```
AuthContext state:                                  AuthContext.js:32
▶ {user: null, authIsReady: false}
⚠ ▶                                                 react_devtools_backend.js:4049
src\pages\home\Home.js
  Line 1:8:  'styles' is defined but never used
no-unused-vars
AuthContext                                         react_devtools_backend.js:4049
state:
▼ {user: Im, authIsReady: true}
    authIsReady: true
  ▶ user: Im {J: Array(0), l: 'AIzaSyAARDxNZTL21…
  ▶ [[Prototype]]: Object
```

When a user is logged in and the user refreshes the page. Initially the user is null. The useEffect hook is fired which will fire the **project.onAuthStateChanged** fucntion which will fire that dispatch which will update the user.

What we actually want is we dont want to render any component until authIsReady is true. We do this by using the AuthContext api inside the App.js component

```jsx
import { useAuthContext } from './hooks/useAuthContext'

function App() {
  const { authIsReady } = useAuthContext()

  return (
    <div className="App">
      {authIsReady && (
        <BrowserRouter>
          <Navbar />
          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <Route path="/login">
              <Login />
            </Route>
            <Route path="/signup">
              <Signup />
            </Route>
          </Switch>
        </BrowserRouter>
      )}
    </div>
  );
```

Route Guarding

If a user is not logged in and they try to go to the home page, the project should protect that home page against them until they're logged in.

OR

If a user is already logged in, he shouldnt go to the login page and login again, we dont want the user to login again when he is already logged in

This is where route huarding comes to play

```
return (
  <div className="App">
    {authIsReady && (
      <BrowserRouter>
        <Navbar />
        <Switch>
          <Route exact path="/">
            {!user && <Redirect to="/login" />}
            {user && <Home />}
          </Route>
          <Route path="/login">
            {user && <Redirect to="/" />}
            {!user && <Login />}
          </Route>
          <Route path="/signup">
            {user && <Redirect to="/" />}
            {!user && <Signup />}
          </Route>
        </Switch>
      </BrowserRouter>
    )}
  </div>
```

App.js

If a user is not logged in -> Redierct to login page

User can access home page if he has already logged in

If a user is already logged in and tries to login again -> Redirect to home page

Login page is shown only if he has not logged in

If a user is already logged in -> Redirect to home page

User can see the signup page if hes not logged in

This also takes care of redirecting the user to home page on login and signup. And redirecting back to login page on logout

useFirestore hook

```
import { useReducer, useEffect, useState } from "react"
import { projectFirestore, timestamp } from "../firebase/config"

let initialState = {
  document: null,
  isPending: false,
  error: null,
  success: null
}

const firestoreReducer = (state, action) => {
  switch (action.type) {
    case 'IS_PENDING':
      return { isPending: true, document: null, success: false, error: null }
    case 'ADDED_DOCUMENT':
      return { isPending: false, document: action.payload, success: true, error: null }
    case 'DELETED_DOCUMENT':
      return { isPending: false, document: null, success: true, error: null }
    case 'ERROR':
      return { isPending: false, document: null, success: false, error: action.payload }
    default:
      return state
  }
}

export const useFirestore = (collection) => {
  const [response, dispatch] = useReducer(firestoreReducer, initialState)
  const [isCancelled, setIsCancelled] = useState(false)

  // collection ref
  const ref = projectFirestore.collection(collection)

  // only dispatch is not cancelled
  const dispatchIfNotCancelled = (action) => {
    if (!isCancelled) {
      dispatch(action)
    }
  }

  // add a document
  const addDocument = async (doc) => {
    dispatch({ type: 'IS_PENDING' })
    try {
      const createdAt = timestamp.fromDate(new Date())
      const addedDocument = await ref.add({ ...doc, createdAt })
      dispatchIfNotCancelled({ type: 'ADDED_DOCUMENT', payload: addedDocument })
    }
    catch (err) {
      dispatchIfNotCancelled({ type: 'ERROR', payload: err.message })
    }
  }

  // delete a document
  const deleteDocument = async (id) => {
    dispatch({ type: 'IS_PENDING' })

    try {
      const deletedDocument = await ref.doc(id).delete()
      dispatchIfNotCancelled({ type: 'DELETED_DOCUMENT' })
    }
    catch (err) {
      dispatchIfNotCancelled({ type: 'ERROR', payload: 'could not delete' })
    }
  }

  useEffect(() => {
    return () => setIsCancelled(true)
  }, [])

  return { addDocument, deleteDocument, response }
}
```

useFirestore.js

```
import firebase from 'firebase/app'
import 'firebase/firestore'
import 'firebase/auth'

const firebaseConfig = {
  apiKey: "AIzaSyAARDxNZTL216xkei9QgMOiSQ410EW-kmw",
  authDomain: "mymoney-c0596.firebaseapp.com",
  projectId: "mymoney-c0596",
  storageBucket: "mymoney-c0596.appspot.com",
  messagingSenderId: "443730170101",
  appId: "1:443730170101:web:0bcbbf2e4a22a5c6fbf5bc"
}

// init firebase
firebase.initializApp(firebaseConfig)

// init services
const projectFirestore = firebase.firestore()
const projectAuth = firebase.auth()

// timestamp
const timestamp = firebase.firestore.Timestamp

export { projectFirestore, projectAuth, timestamp }
```

firebaseConfig.js

Timestamp can be used to sort the data according to time

using useFirestore hook

```
import { useAuthContext } from '../../hooks/useAuthContext'
import { useFirestore } from '../../hooks/useFirestore'
const { addDocument, response } = useFirestore('transactions')
const { user } = useAuthContext()

addDocument({
  user.uid,
  name,
  amount
})
```

This is used to show content specific to user only

useCollection hook

```
import { useEffect, useState } from "react"
import { projectFirestore } from "../firebase/config"

export const useCollection = (collection, _query) => {
  const [documents, setDocuments] = useState(null)
  const [error, setError] = useState(null)

  const query = useRef(_query).current
  const orderBy = useRef(_orderBy).current
  useEffect(() => {
    let ref = projectFirestore.collection(collection)

    if (query) {
      ref = ref.where(...query)
    }
    if (orderBy) {
      ref = ref.orderBy(...orderBy)
    }

    const unsubscribe = ref.onSnapshot((snapshot) => {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({ ...doc.data(), id: doc.id })
      })

      // update state
      setDocuments(results)
      setError(null)
    }, (error) => {
      console.log(error)
      setError('could not fetch the data')
    })

    // unsubscribe on unmount
    return () => unsubscribe()

  }, [collection, query, orderBy])

  return { documents, error }
}
```

We are doing this because _query is an array and is a reference type and we have seen earlier how using reference types as dependency causes problems.
Which is why we use the useRef

Used to fetch data from firestore

```
const { documents, error } = useCollection(
  'transactions',
  ["uid", "==", user.uid],
  ["createdAt", "desc"]
)
```

Firebase query to fetch transactions specific to user

Order transactions by 'createdAt' property in descending order

useDocument hook

```
import { useEffect, useState } from "react"
import { projectFirestore } from "../firebase/config"

export default function useDocument(collection, id) {
  const [document, setDocument] = useState(null)
  const [error, setError] = useState(null)

  // realtime document data
    useEffect(() => {
        const ref = projectFirestore.collection(collection).doc(id)

        const unsubscribe = ref.onSnapshot(snapshot => {
            // need to make sure the doc exists & has data
            if(snapshot.data()) {
                setDocument({...snapshot.data(), id: snapshot.id})
                setError(null)
            }
            else {
                setError('No such document exists')
            }
        }, err => {
            console.log(err.message)
            setError('failed to get document')
        })

        // unsubscribe on unmount
        return () => unsubscribe()

    }, [collection, id])

  return { document, error }      You, 5 minutes ago • Project discr
}
```

Handling file input

```jsx
<label>
  <span>profile thumbnail:</span>
  <input
    required
    type="file"
    onChange={handleFileChange}
  />
</label>
```

```jsx
const [thumbnail, setThumbnail] = useState(null)
const [thumbnailError, setThumbnailError] = useState(null)

const handleFileChange = (e) => {
  setThumbnail(null)
  let selected = e.target.files[0]
  console.log(selected)

  if (!selected) {
    setThumbnailError('Please select a file')
    return
  }
  if (!selected.type.includes('image')) {
    setThumbnailError('Selected file must be an image')
    return
  }
  if (selected.size > 100000) {
    setThumbnailError('Image file size must be less than 100kb')
    return
  }

  setThumbnailError(null)
  setThumbnail(selected)
  console.log('thumbnail updated')

}
```

```
react_devtools_backend.js:4049
File {name: 'yoshi.png', lastModified: 1628435570615, lastMo
▼difiedDate: Sun Aug 08 2021 16:12:50 GMT+0100 (British Summe
r Time), webkitRelativePath: '', size: 7876, …}
  lastModified: 1628435570615
  ▶lastModifiedDate: Sun Aug 08 2021 16:12:50 GMT+0100 (Brit…
  name: "yoshi.png"
  size: 7876
  type: "image/png"
  webkitRelativePath: ""
  ▶[[Prototype]]: File
>
```

React Select

1. npm install react-select
2. `import Select from 'react-select`
3. Array of options

```
const categories = [
  { value: 'development', label: 'Development' },
  { value: 'design', label: 'Design' },
  { value: 'sales', label: 'Sales' },
  { value: 'marketing', label: 'Marketing' },
]
```

4. Select form field

```
<label>
  <span>Project category:</span>
  <Select
    onChange={(option) => setCategory(option)}
    options={categories}
  />
</label>
```

```
<Select
  onChange={(option) => setAssignedUsers(option)}
  options={users}
  isMulti
/>
```

Allows us to select multiple options

OUTPUT

We cannot add **required** to select tag like other input fields, we have to explicitly check if any option is selected or not

```
if (!category) {
    dispatch({ type: 'SET_FORM_ERROR', error: 'Please select a project category.' });
    return;       You, 21 seconds ago • Uncommitted changes
}
if (assignedUsers.length < 1) {
    dispatch({ type: 'SET_FORM_ERROR', error: 'Please assign the project to at least 1 user' });
    return;
}
```

Using firebase storage

1. Firebase storage setup in config.js

```
import 'firebase/storage'
```

```
// init services
const projectFirestore = firebase.firestore()
const projectAuth = firebase.auth()
const projectStorage = firebase.storage()

// timestamp
const timestamp = firebase.firestore.Timestamp

export { projectFirestore, projectAuth, projectStorage, timestamp }
```

2. Updating profile photo in useSignup hook

```
const signup = async (email, password, displayName, thumbnail) => {
  setError(null)
  setIsPending(true)

  try {
    // signup
    const res = await projectAuth.createUserWithEmailAndPassword(email, password)

    if (!res) {
      throw new Error('Could not complete signup')
    }

    // upload user thumbnail
    const uploadPath = `thumbnails/${res.user.uid}/${thumbnail.name}`
    const img = await projectStorage.ref(uploadPath).put(thumbnail)
    const imgUrl = await img.ref.getDownloadURL()

    // add display name to user
    await res.user.updateProfile({ displayName, photoURL: imgUrl })

    // dispatch login action
    dispatch({ type: 'LOGIN', payload: res.user })
```

Uploading pic to storage and getting its url

Uploading profile of user with userName and profile photo

Firestore Rules

Firestore Database


```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2021, 10, 27);
    }
  }
}
```

version of rules we are using

service for which we are writing rules

Base path of the database. {} represents wildcard ie this means this can be any database name

1. The `match /databases/{database}/documents` line indicates that these rules apply to all documents within the Firestore database.
2. The rule `match /{document=**}` applies to any document within the database, regardless of its path.
3. The `allow read, write: if request.time < timestamp.date(2023, 6, 29);` line specifies the access conditions for reading and writing data. In this case, it allows anyone to read and write data until June 29, 2023. After that date, all client requests to the Firestore database will be denied until the security rules are updated.

The `match` keyword in Firestore security rules is used to define the scope of the rules. It determines which documents or collections the rules will apply to. The `match` keyword is followed by a path pattern that specifies the location of the documents or collections within the database.

The general syntax for using the `match` keyword is as follows:

rust — Copy code

```
match /path_pattern {
  // security rules
}
```

Here, `path_pattern` represents the path within the Firestore database where the rules should be applied. It can contain placeholders or wildcards to match multiple documents or collections. The path pattern follows the same structure as Firestore document and collection paths.

```
match /recipes/{document=**} {
  allow read, write: if request.time < timestamp.date(2023, 6, 29);
}
```

We can update the above rule like this. this says that all documents in the recipes collection can be read or written before 29/06/2023

Firebase CLI

Instead of editing the rules directly in firestore app, we can write the rules in VS code and then host it to the firestore app

```
npm install -g firebase-tools
```

install firebase-tools

```
firebase login
```

This will open the browser to log in locally to the firebase

```
firebase init
```

will create a new firebase project on the frontend

Now it will ask if we want to create a new firebase project or connect to an existing one.
If we want to create a new project, we can or we can select from our existing projects

```
You're about to initialize a Firebase project in this directory:

  C:\Users\Shaun\Documents\Tuts\React-Firebase\my-money

? Are you ready to proceed? Yes
? Which Firebase features do you want to set up for this directory? Press Space to select features, then
Enter to confirm your choices.
default instance
 (*) Firestore: Configure security rules and indexes files for Firestore
 ( ) Functions: Configure a Cloud Functions directory and its files
>(*) Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
 ( ) Hosting: Set up GitHub Action deploys
 ( ) Storage: Configure a security rules file for Cloud Storage
 ( ) Emulators: Set up local emulators for Firebase products
(Move up and down to reveal more choices)
```

Will ask you for the features required. Move up and down and press space to select. Select **Firestore** and **Hosting**

```
? What file should be used for Firestore Rules? firestore.rules
```

Will ask for name of the firestore rules file. Just hit enter ie default file name

```
? What file should be used for Firestore indexes? firestore.indexes.json
```

Just hit enter for default file name

```
? What do you want to use as your public directory? build
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? Set up automatic builds and deploys with GitHub? No
```

But when it comes to naming the public directory, do not enter because we need to change it. So what it's asking about this public directory is that, it means the directory that Firebase is going to deploy later when we want to host our website. Now, the default folder name given here is public, but we already use that for public files in our React application.
Now we want this to be called build instead, because when we like to build our React application, React puts all of our build files into the build folder and that is the folder we want to deploy later

Lists firebase project id for the application

Config file for project. It tells firebase where to find the rules file and anyother firebase files we have locally

For database indexes

Firestore rules

Securing Data Collection

In the finance tracker project we created earlier, the transaction are listed according to the user who has logged in. But the user can list the transactions of other users as well by slightly manipulating the code and update and delete other users data. So we need to secure other users data.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /transactions/{document=**} {

      // only auth
      allow create: if request.auth != null;

      // logged in user uid must match the document creator to read & delete
      allow read, delete: if request.auth.uid == resource.data.uid;
    }
  }
}
```

```
// only auth
allow create: if request.auth != null;
```

___ Allow only authenticated users to create new documents(transactions)

Remember when we talked about how authentication works, I said that the user's authentication token will be sent to the server on every request it makes. So if someone makes a request to the database, it gets sent to the server and Firebase can validate that token to see if the user is authenticated.

Now inside Firestore rules, we get access to that user auth status and we can use it as a condition.

If a user is logged in , the **auth property** is not equal to null
If user is not logged in, the **auth** property is null

```
// logged in user uid must match the document creator to read & delete
allow read, delete: if request.auth.uid == resource.data.uid;
```

user can read and delete only those documents (transaction) that belong to him

`request.auth.uid` — uid of user that is making request

`resource.data.uid;` — resource is nothing but the document we are trying to fetch. We can use the **.data** property to access its data

Delpoying

```
firebase deploy --only firestore
```

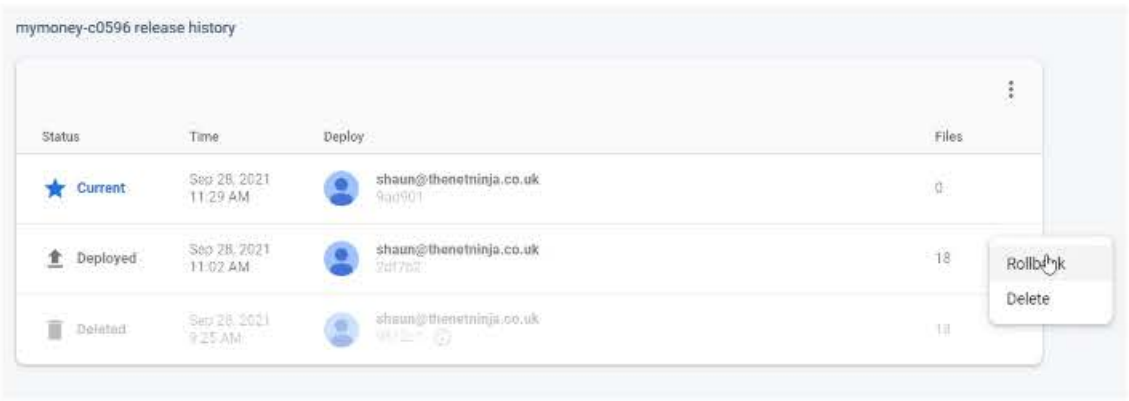
Deployin React App

Deploying on Firebase

1. npm run build
2. firebase deploy

Go to the hosting tab in firebase to see the project

Updating the site &Redeploying ← After updating run the same commands



When we redeploy multiple times, we see the list of deployments
How to go to some previous deployments?
We just right click on the deployment and select rollback

Deploying on github

1. npm install gh-pages --save-dev
2. Make the following changes in the package.json

```
{
  "homepage": "https://github.com/shivkaraman/myMoney-Expnese-Tracker/",
  "name": "14_my-money",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "firebase": "8.5",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.11.2",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  Debug
  "scripts": {
    "start": "react-scripts start",
    "predeploy" : "npm run build",
    "deploy" : "gh-pages -d build",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

3. Commit all the changes and run -> npm run deploy

React Router Version 6

```jsx
<BrowserRouter>
  <nav>
    <h1>The Ninja Clothing Company</h1>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
    <Link to="/products">Products</Link>
  </nav>
  <Switch>
    <Route exact path="/" component={Home} />
    <Route path="/about">
      <About />
    </Route>
    <Route path="/products/:id">
      <ProductDetails />
    </Route>
    <Route path="/products">
      <Products />
    </Route>
  </Switch>
</BrowserRouter>
```

⇨

```jsx
<BrowserRouter>
  <nav>
    <h1>The Ninja Clothing Company</h1>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
    <Link to="/products">Products</Link>
  </nav>
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="/products/:id" element={<ProductDetails />} />
    <Route path="/products" element={<Products />} />
  </Routes>
</BrowserRouter>
```

1. Switch statement is no longer used in version 6 and instead we wrap all of our different routes in the Routes component.
2. Each root component should have an element prop where we output the JSX template we want to render whenva user visits that particular path.
3. And finally, we don't need the exact prop for exact matches anymore, because that's now the default behavior of version 6.

The JSX component inside the element can be any component we inported or can be just an inlinne JSX template which will be rendered

```jsx
<Route path="/test" element={(
  <div>
    <h2>Test Page</h2>
    <p>hello!</p>
  </div>
)} />
```

⇨

Test Page

hello!

Redirects & useNavigate

```jsx
<Route path="/redirect" element={<Redirect to="/about" />} />
```

⇩

```jsx
<Route path="/redirect" element={<Navigate to="/about" />} />
```

Programatic redirects : In version 5, we used the useHistory and History.push(route) for programatic redirects. In version 6, we no longer use it. We use useNavigate hook instead

```jsx
const navigate = useNavigate()
```

```jsx
<button onClick={() => navigate('/products')}>See our products</button>
```

1. Package Import:

   In React Router v6, the package names have changed. Instead of importing from `react-router-dom`, you now import from `react-router`. The components are directly exported from the package.

```jsx
// React Router v5
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

// React Router v6
import { Router, Route, Switch } from 'react-router';
```

2. Route Configuration:

   In version 5, you use the `component` prop to specify the component to render for a route. In version 6, you use the `element` prop to specify the JSX element to render.

```jsx
// React Router v5
<Route path="/about" component={About} />

// React Router v6
<Route path="/about" element={<About />} />
```

3. Nesting Routes:

   In React Router v6, nesting routes is more straightforward and intuitive. Instead of using the `Switch` component, you can nest routes directly within parent routes.

```jsx
// React Router v5
<Switch>
  <Route path="/dashboard" component={Dashboard} />
  <Route path="/dashboard/profile" component={UserProfile} />
</Switch>

// React Router v6
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<UserProfile />} />
</Route>
```

4. Route Parameters:

   In React Router v6, route parameters are defined using the `:` syntax within the `path` prop.

```jsx
// React Router v5
<Route path="/users/:id" component={UserDetail} />

// React Router v6
<Route path="/users/:id" element={<UserDetail />} />
```

5. Route Rendering:

   In version 6, you can use the `useRoutes` hook to define routes within a component and render the matched route using the `Outlet` component.

```jsx
// React Router v6
import { useRoutes, Outlet } from 'react-router';

function App() {
  const routes = useRoutes([
    { path: '/', element: <Home /> },
    { path: '/about', element: <About /> },
    { path: '/users/:id', element: <UserDetail /> },
  ]);

  return (
    <div>
      {routes}
      <Outlet />
    </div>
  );
}
```

Firebase Version 9

```
import firebase from 'firebase/app'
import 'firebase/firestore'
import 'firebase/auth'

const firebaseConfig = {
  apiKey: "AIzaSyAARDxNZTL216xkei9QgMOiSQ410EW-kmw",
  authDomain: "mymoney-c0596.firebaseapp.com",
  projectId: "mymoney-c0596",
  storageBucket: "mymoney-c0596.appspot.com",
  messagingSenderId: "443730170101",
  appId: "1:443730170101:web:0bcbbf2e4a22a5c6fbf5bc"
}

// init firebase
firebase.initializeApp(firebaseConfig)

// init services
const projectFirestore = firebase.firestore()
const projectAuth = firebase.auth()

export { projectFirestore, projectAuth }
```

```
import { initializeApp } from 'firebase/app'
import { getFirestore } from 'firebase/firestore'
import { getAuth } from 'firebase/auth'

const firebaseConfig = {
  apiKey: "AIzaSyBFPurFyXbqU4T9XegQmKszJxjbIGMX624",
  authDomain: "readinglistapp-da356.firebaseapp.com",
  projectId: "readinglistapp-da356",
  storageBucket: "readinglistapp-da356.appspot.com",
  messagingSenderId: "26365571782",
  appId: "1:26365571782:web:97f131332cbb7c29b35dc7"
}

// init firebase
initializeApp(firebaseConfig)

// init firestore
const db = getFirestore()

// init firebase auth
const auth = getAuth()

export { db, auth }
```

Fetching a collection

```
import { projectFirestore } from '../../firebase/config'

useEffect(() => {
  setIsPending(true)

  projectFirestore.collection('recipes').get().then((snapshot) => {
    if (snapshot.empty) {
      setError('No recipes to load')
      setIsPending(false)
    } else {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({ id: doc.id, ...doc.data() })
      })
      setData(results)
      setIsPending(false)
    }
  }).catch(err => {
    setError(err.message)
    setIsPending(false)
  })

}, []);
```

```
import { db } from '../firebase/config'
import { collection, getDocs } from 'firebase/firestore'

useEffect(() => {
  const ref = collection(db, 'books')

  getDocs(ref)
    .then((snapshot) => {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({id: doc.id, ...doc.data()})
      })
      setBooks(results)
    })
}, [])
```

Name of collection

Real time collection data

```
useEffect(() => {
  setIsPending(true)

  const unsub = projectFirestore.collection('recipes').onSnapshot((snapshot) => {
    if (snapshot.empty) {
      setError('No recipes to load')
      setIsPending(false)
    } else {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({ id: doc.id, ...doc.data() })
      })
      setData(results)
      setIsPending(false)
    }
  }, (err) => {
    setError(err.message)
    setIsPending(false)
  })

  return () => unsub()
}, [])
```

CLEANUP FUNC

Home.js

```
import { useState, useEffect } from 'react'
import { db } from '../firebase/config'

// firebase imports
import { collection, onSnapshot } from 'firebase/firestore'

export const useCollection = (c) => {
  const [documents, setDocuments] = useState(null)

  useEffect(() => {
    let ref = collection(db, c)

    const unsub = onSnapshot(ref, (snapshot) => {
      let results = []
      snapshot.docs.forEach(doc => {
        results.push({...doc.data(), id: doc.id})
      })
      setDocuments(results)
    })

    return () => unsub()
  }, [c])

  return { documents }
}
```

Fetching a document

```
import { useEffect, useState } from "react"
import { projectFirestore } from "../firebase/config"

export default function useDocument(collection, id) {
  const [document, setDocument] = useState(null)
  const [error, setError] = useState(null)

  // realtime document data
    useEffect(() => {
        const ref = projectFirestore.collection(collection).doc(id)

        ref.get().then(doc => {
            // need to make sure the doc exists & has data
            if(doc.data()) {
                setDocument({...doc.data(), id: doc.id})
                setError(null)
            }
            else {
                setError('No such document exists')
            }
        }, err => {
            console.log(err.message)
            setError('failed to get document')
        })

    }, [collection, id])

  return { document, error }
}
```

```
import { useEffect, useState } from "react"
import { useEffect, useState } from 'react';
import { doc, getDoc } from 'firebase/firestore';
import { projectFirestore } from '../firebase/config';

export default function useDocument(collection, id) {
  const [document, setDocument] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    const ref = doc(projectFirestore, collection, id);

    getDoc(ref)
      .then((docSnapshot) => {
        if (docSnapshot.exists()) {
          setDocument({ ...docSnapshot.data(), id: docSnapshot.id });
          setError(null);
        } else {
          setError('No such document exists');
        }
      })
      .catch((err) => {
        console.log(err.message);
        setError('Failed to get document');
      });

  }, [collection, id]);

  return { document, error };
}
```

Realtime document data

```
import { useEffect, useState } from "react"
import { projectFirestore } from "../firebase/config"

export default function useDocument(collection, id) {
  const [document, setDocument] = useState(null)
  const [error, setError] = useState(null)

  // realtime document data
    useEffect(() => {
        const ref = projectFirestore.collection(collection).doc(id)

        const unsubscribe = ref.onSnapshot(snapshot => {
            // need to make sure the doc exists & has data
            if(snapshot.data()) {
                setDocument({...snapshot.data(), id: snapshot.id})
                setError(null)
            }
            else {
                setError('No such document exists')
            }
        }, err => {
            console.log(err.message)
            setError('failed to get document')
        })

        // unsubscribe on unmount
        return () => unsubscribe()

    }, [collection, id])

  return { document, error }
}
```

You, 2 weeks ago • Project discription page added …

```
import { useEffect, useState } from 'react';
import { collection, doc, onSnapshot } from 'firebase/firestore';
import { projectFirestore } from './firebase'; // Replace './firebase' with the actual path to your Firebase configuration file

export default function useDocument(collection, id) {
  const [document, setDocument] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    const ref = doc(projectFirestore, collection, id);

    const unsubscribe = onSnapshot(ref, (snapshot) => {
      if (snapshot.exists()) {
        setDocument({ ...snapshot.data(), id: snapshot.id });
        setError(null);
      } else {
        setError('No such document exists');
      }
    }, (err) => {
      console.log(err.message);
      setError('Failed to get document');
    });

    return () => unsubscribe();
  }, [collection, id]);

  return { document, error };
}
```

Adding data

```
const doc = { title, ingredients, method, cookingTime: cookingTime + ' minutes' }

try {
  await projectFirestore.collection('recipes').add(doc)
  history.push('/')
} catch(err) {
  console.log(err)
}
```

```
const [newBook, setNewBook] = useState('')

const handleSubmit = async (e) => {
  e.preventDefault()

  const ref = collection(db, 'books')

  await addDoc(ref, {
    title: newBook
  })

  setNewBook('')
}
```

Deleting data

```
projectFirestore.collection('recipes').doc(id).delete()
```

```
const ref = doc(db, 'books', id)
await deleteDoc(ref)
```

Signup

```
import { useState } from 'react'
import { projectAuth } from '../firebase/config'

export const useSignup = () => {
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)

  const signup = async (email, password, displayName) => {
    setError(null)
    setIsPending(true)

    try {
      // signup user
      const res = await projectAuth.createUserWithEmailAndPassword(email, password)
      console.log(res.user)

      if (!res) {
        throw new Error('Could not complete signup')
      }

      // add display name to user
      await res.user.updateProfile({ displayName })

      setIsPending(false)
      setError(null)

    }
    catch (err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
    }
  }

  return { error, isPending, signup }
}
```

```
import { useState } from 'react'

// firebase imports
import { auth } from '../firebase/config'
import { createUserWithEmailAndPassword } from "firebase/auth"

export const useSignup = () => {
  const [error, setError] = useState(null)
        const signup: (email: any, password: any) => void
  const signup = (email, password) => {
    setError(null)
    createUserWithEmailAndPassword(auth, email, password)
      .then((res) => {
        console.log('user signed up:', res.user)
      })
      .catch((err) => {
        setError(err.message)
      })

  }

  return { error, signup }
}
```

Logout

```
import { useState } from 'react'
import { projectAuth } from '../firebase/config'
import { useAuthContext } from './useAuthContext'

export const useLogout = () => {
  const [error, setError] = useState(null)
  const [isPending, setIsPending] = useState(false)

  const logout = async () => {
    setError(null)
    setIsPending(true)

    // sign the user out
    try {
      await projectAuth.signOut()

      setIsPending(false)
      setError(null)
    }
    catch(err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
    }
  }

  return { logout, error, isPending }

}
```

useLogout.js

```
// firebase imports
import { auth } from '../firebase/config'
import { signOut } from 'firebase/auth'

export const useLogout = () => {

  const logout = () => {
    signOut(auth)
      .then(() => {
        console.log('user signed out')
      })
      .catch((err) => {
        console.log(err.message)
      })
  }

  return { logout }

}
```

Login

```
const login = async (email, password) => {
  setError(null)
  setIsPending(true)

  // sign the user out
  try {
    const res = await projectAuth.signInWithEmailAndPassword(email, password)

      setIsPending(false)
      setError(null)
    }
  }
  catch(err) {
      console.log(err.message)
      setError(err.message)
      setIsPending(false)
  }
}
```

```
import { useState } from 'react'

// firebase imports
import { auth } from '../firebase/config'
import { signInWithEmailAndPassword } from "firebase/auth"

export const useLogin = () => {
  const [error, setError] = useState(null)

  const login = (email, password) => {
    setError(null)
    signInWithEmailAndPassword(auth, email, password)
      .then((res) => {
        console.log('user logged in:', res.user)
      })
      .catch((err) => {
        setError(err.message)
      })

  }

  return { error, login }
}
```

BY SHIVKARAMAN