B2V Technology

React Learning Beginner's

www.b2vtech.com

Table of Content

Environment Setup

Step 1: Install Node.js installer for windows.Once downloaded open Node JS without disturbing other settings, click on the **Next** button until it's completely installed.

Step 2: Open command prompt to check whether it is completely installed and check version of node.

Step 3: Now in the terminal run the below command:

```
npm install -g create-react-app
```

Step 4: Now Create a new folder where you want to make your react app using the below command:

```
npx create-react-app Folder_Name
```

Step 5: Navigate to Your Project Directory

```
cd  Folder_Name
```

Step 6: Start the Development Server

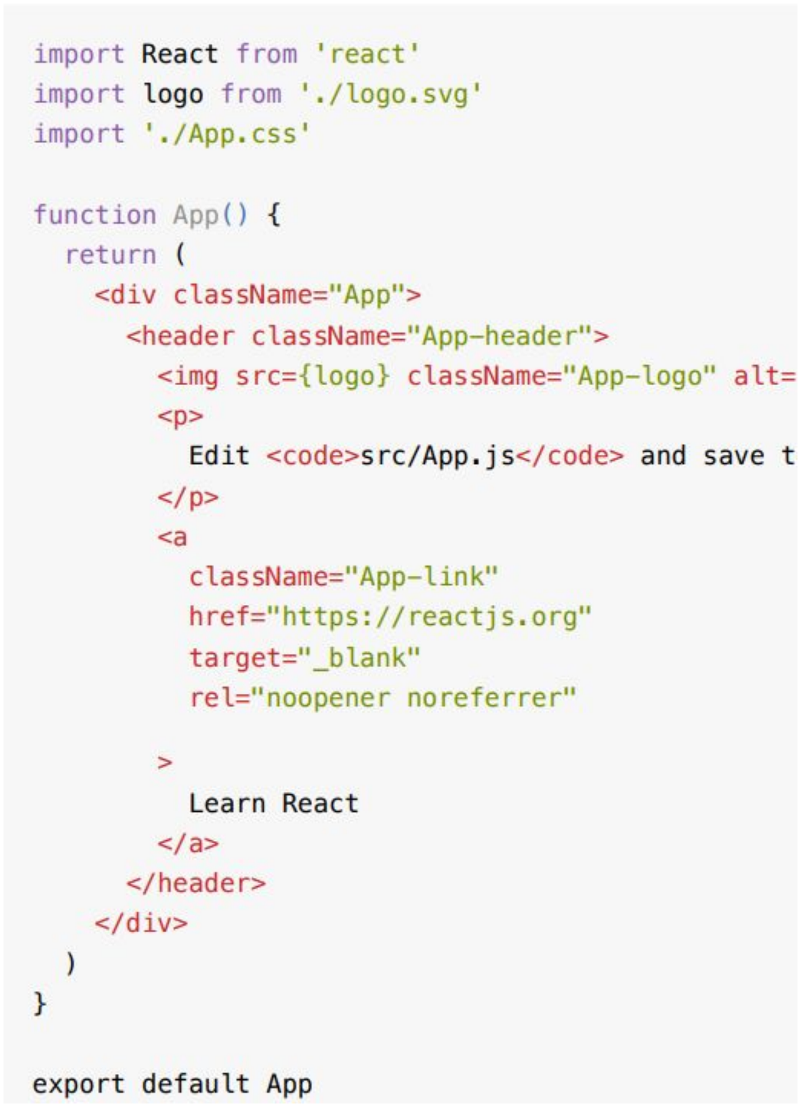
```
npm start
```

Components/Reusable Components

You just saw how to create your first React application.

This application comes with a series of files that do various things, mostly related to configuration, but there's one file that stands out: App.js .
App.js is the first React Component you meet.

Its code is this:



```
import React from 'react'
import logo from './logo.svg'
import './App.css'

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt=
        <p>
          Edit <code>src/App.js</code> and save t
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"

        >
          Learn React
        </a>
      </header>
    </div>
  )
}

export default App
```

What is Reusable component?

Reusable components allow you to use an element in several parts of your API, without having to define it multiple times. If you need to reuse the same header in ten different operations, you can create a component, and then refer to the component in all the operations where it is needed.

Single Responsibility Principle (SRP)

This principle states that a class or component should have a single responsibility or single reason to change. By keeping components focused on a specific task, you improve code readability, maintainability, and reusability.

It promotes breaking down complex functionality into smaller, focused parts that are easier to understand, test, and maintain.

It encourages components to have clear and specific responsibilities, enhancing their reusability and maintainability.

It helps in avoiding tightly coupled components by keeping them focused on specific tasks.

By carefully structuring our components, adhering to the Single Responsibility Principle, and embracing concepts like Atomic Design and Component Composition, we can create code that is more modular, easier to test, and simpler to maintain.

This approach leads to a more efficient development process and ultimately results in high-quality, scalable React applications.

JSX (Javascript with XML)

As we learned earlier, React JSX is an extension to JavaScript. It enables developer to create virtual DOM using XML syntax. It compiles down to pure JavaScript (React.createElement function calls). Since it compiles to JavaScript, it can be used inside any valid JavaScript code. For example, below codes are perfectly valid.

Assign to a variable:

```
var greeting = <h1>Hello React!</h1>
```

Assign to a variable based on a condition:

```
var canGreet = true;
if(canGreet) { greeting =
<h1>Hello React!</h1>
}
```

Can be used as return value of a function.

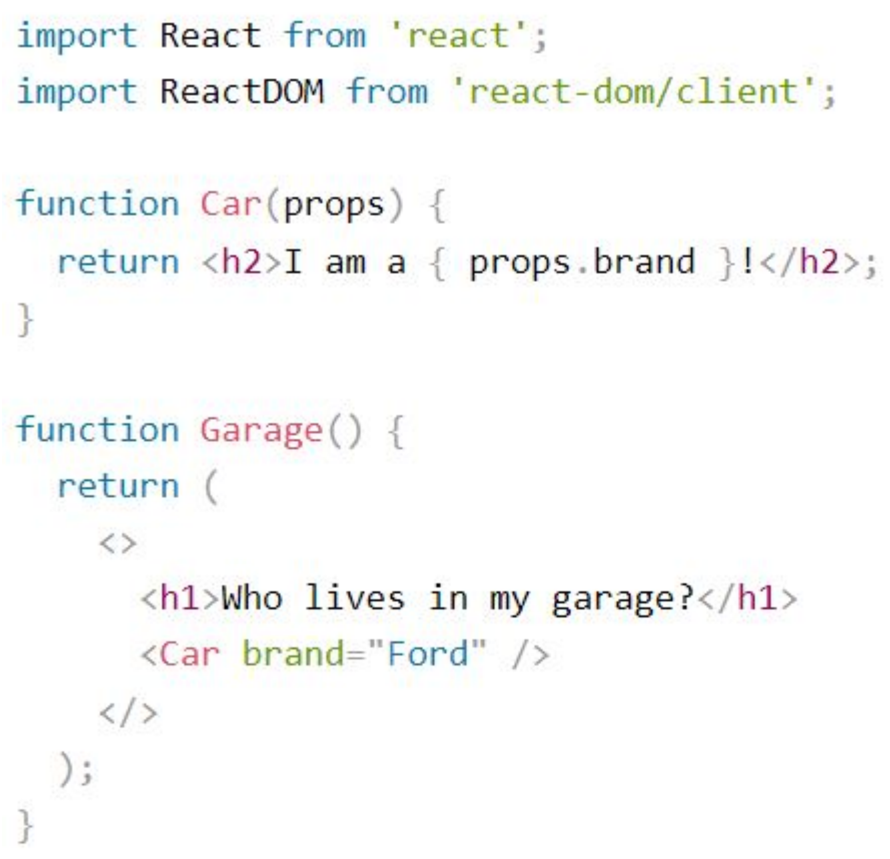
```
function Greeting() { return <h1>Hello React!</h1>}
greeting = Greeting()
```

Can be used as argument of a function.

```
function Greet(message) { ReactDOM.render(message,
document.getElementById('react-app') } Greet(
<h1>Hello React!</h1>}
)
```

Props

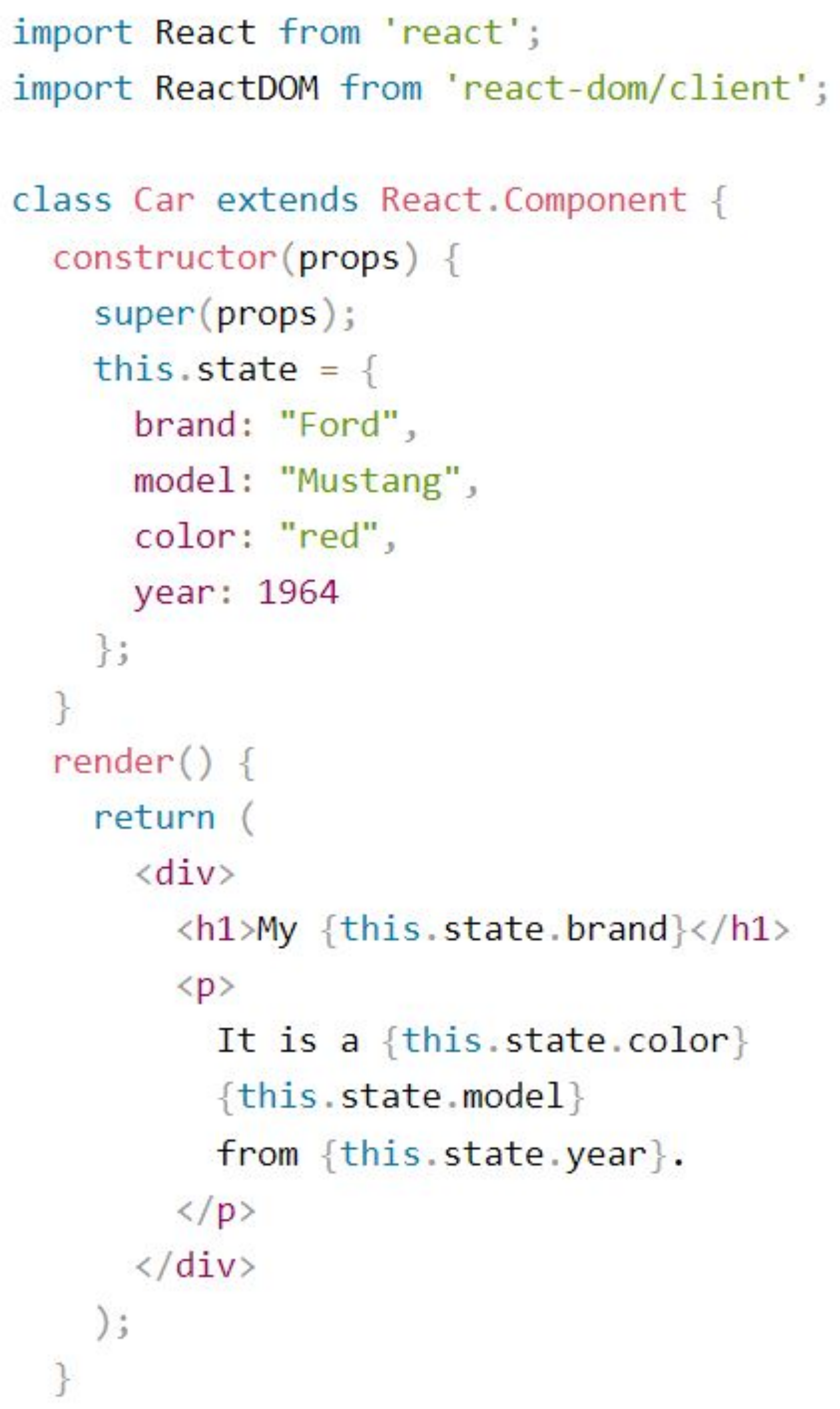
Props are arguments passed into React components.
Props are passed to components via HTML attributes.
props stands for properties.
React Props are like function arguments in JavaScript *and* attributes in HTML.
Note: React Props are read-only! You will get an error if you try to change their value.



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

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}
```

State

React components has a built-in state object.
The state object is where you store property values that belong to the component.
When the state object changes, the component re-renders.



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

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
```

Hooks

Although Hooks generally replace class components, there are no plans to remove classes from React.

What is Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.

Hook Rules

here are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Hook List:

➔ useState
➔ useEffect
➔ useContext
➔ useRef
➔ useReducer
➔ useCallback
➔ useMemo

What is useState hook in React?

useState() allows one to declare a state variable inside a function. It should be noted that one use of useState() can only be used to declare one state variable. It was introduced in version 16.8.

Reason to choose useState hook

Functional components are some of the more commonly used components in ReactJS. Most developers prefer using functional components over class-based components for the simple reason that functional components require less coding (on the developer's part). However, two main features for the class are lost when one goes with a functional component – a dedicated state that persists through render calls as well as the use of lifecycle functions to control how the component looks and behaves at separate stages of its lifecycle.

Refer link : https://react.dev/reference/react/useState

What is useEffect hook in React?

The useEffect hook in React is used to handle the side effects in React such as fetching data, and updating DOM. This hook runs on every render but there is also a way of using a dependency array using which we can control the effect of rendering

Refer link : https://react.dev/reference/react/useEffect

All Hooks Refer link:

https://react.dev/learn
https://react.dev/reference/react
https://www.geeksforgeeks.org/reactjs-hooks/

Conditional Renderings

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.

Logical AND operator (&&)

Another common shortcut you'll encounter is the JavaScript logical AND (&&) operator. Inside React components, it often comes up when you want to render some JSX when the condition is true, **or render nothing otherwise.** With &&, you could conditionally render the checkmark only if isPacked is true:

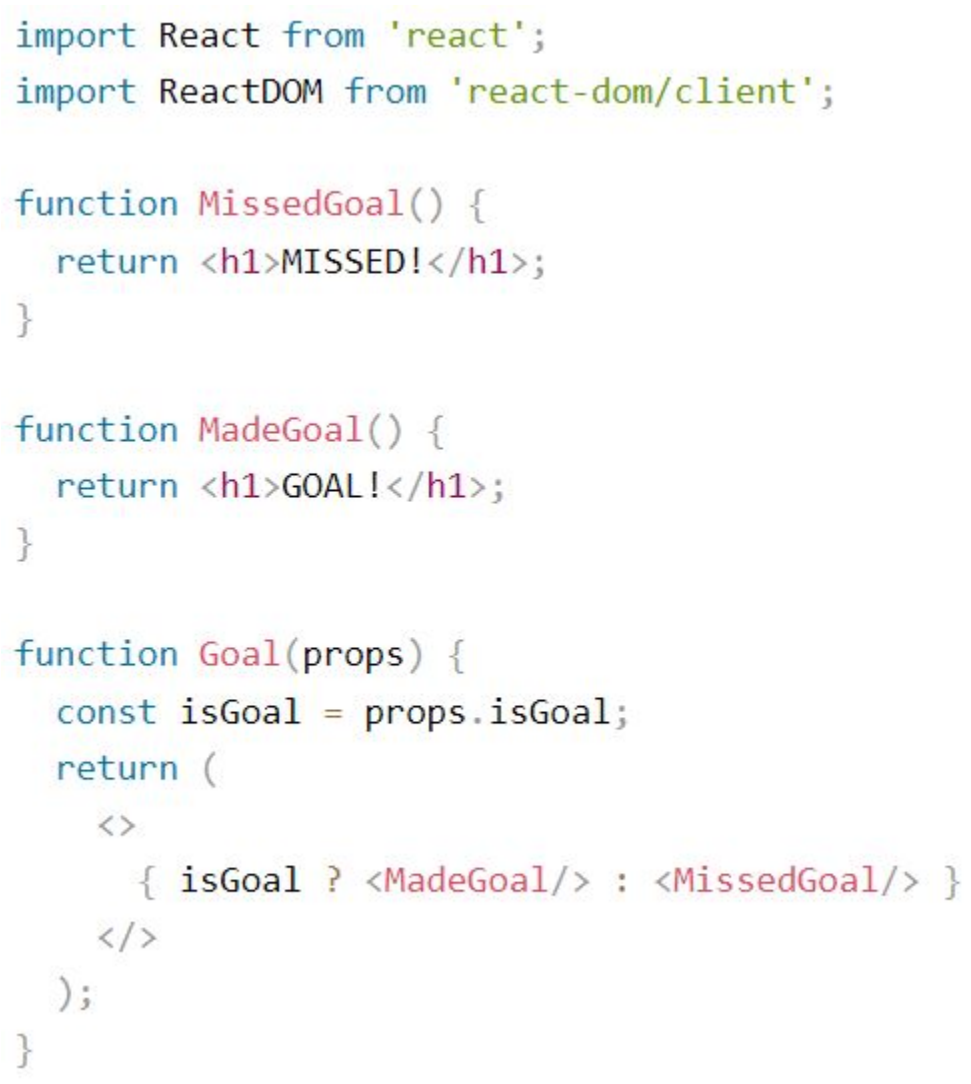
```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}

const cars = ['Ford', 'BMW', 'Audi'];
```

Conditional (ternary) operator (? :)

JavaScript has a compact syntax for writing a conditional expression — the conditional operator or “ternary operator”.

Another way to conditionally render elements is by using a ternary operator.

condition ? true : false



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

function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  return (
    <>
      { isGoal ? <MadeGoal/> : <MissedGoal/> }
    </>
  );
}
```

List and Keys

Whenever you are rendering a list of React components, each component needs to have a key attribute. The key can be any value, but it does need to be unique to that list.

When React has to render changes on a list of items, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference. If there are no keys set for the children, React scans each child. Otherwise, React compares the keys to know which were added or removed from the list

Basic Example

For a class-less React component:

```
function SomeComponent(props){
const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => {item}); }
            return (
                <ul>{getItemsList()}
                </ul>
            );
        }
```

For this example, the above component resolves to:

```
<ul>
    <li>cat</li>
    <li>dog</li>
    <li>rat</li>
</ul>
```

Higher Order Components

Higher Order Components ("HOC" in short) is a react application design pattern that is used to enhance components with reusable code. They enable to add functionality and behaviors to existing component classes.

A HOC is a pure javascript function that accepts a component as it's argument and returns a new component with the extended functionality.

Simple Higher Order Component

Let's say we want to console.log each time the component mounts

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

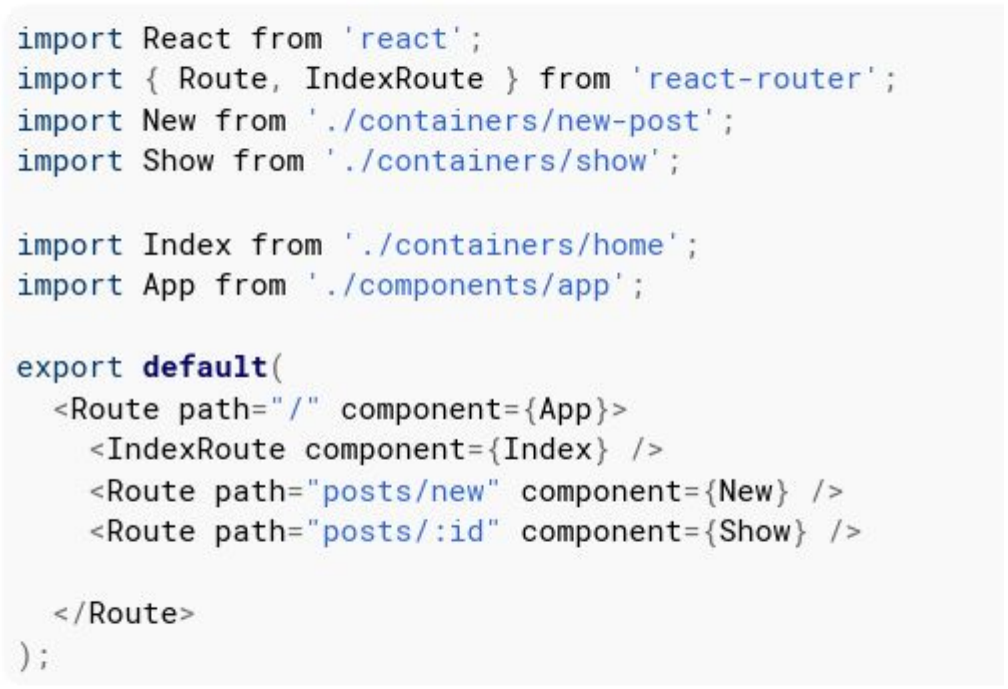
MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component gets logged to console on each mount.
            </div>
        );
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

React Router

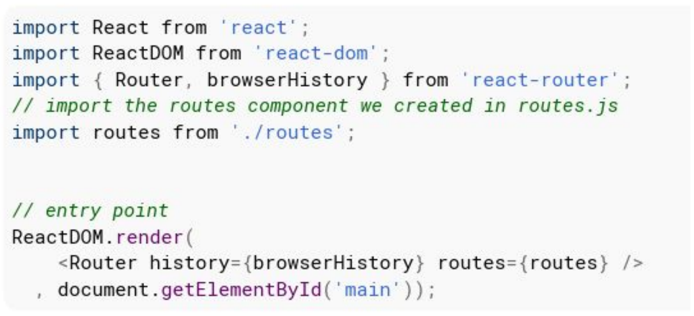
Example Routes.js file, followed by use of Router Link in component

Place a file like the following in your top level directory. It defines which components to render for which paths



```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Now in your top level index.js that is your entry point to the app, you need only render this Router component like so:



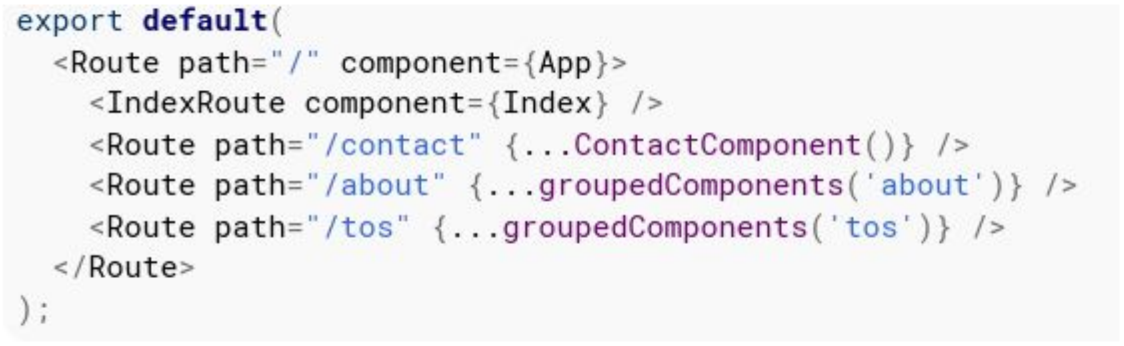
```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Now it is simply a matter of using Link instead of tags throughout your application. Using Link will communicate with React Router to change the React Router route to the specified link, which will in turn render the correct component as defined in routes.js

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
```



```
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Context API

Managing state is an essential part of developing applications in React. A common way to manage state is by passing props. Passing props means sending data from one component to another. It's a good way to make sure that data gets to the right place in a React application.

The Problem with Passing Props

In React, passing props is a fundamental concept that enables a parent component to share data with its child components as well as other components within an application.

In many cases, passing props can be an effective way to share data between different parts of your application. But passing props down a chain of multiple components to reach a specific component can make your code overly cumbersome.

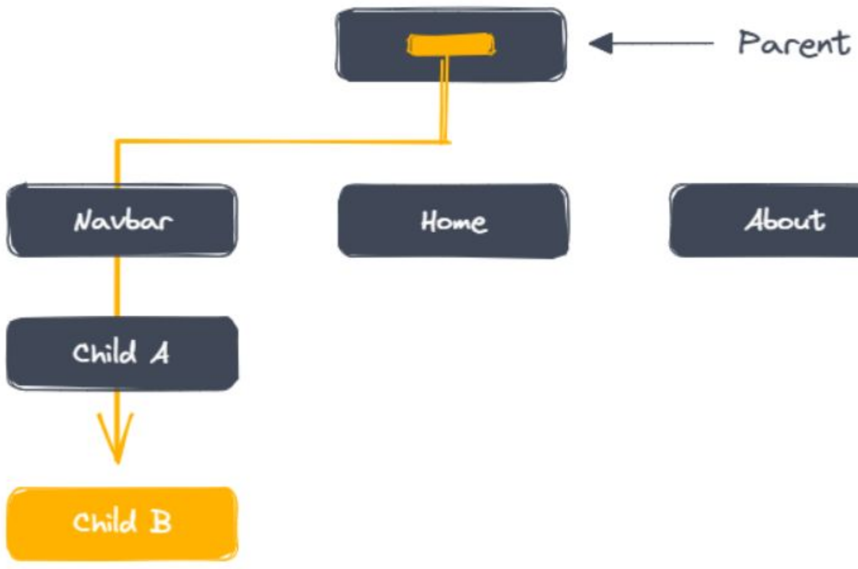


Illustration of passing props from parent to children

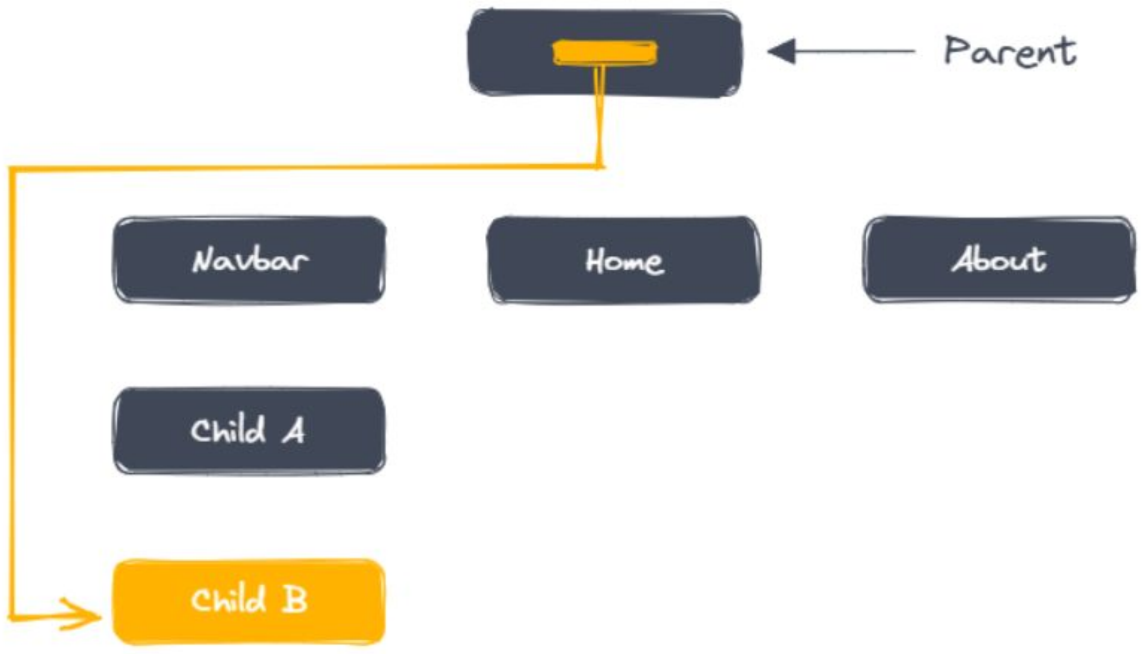
From the above diagram, to pass data down to the component "Child B", we need to pass it down through all the intermediate components, even if those components don't actually use the data themselves. This is what is referred to as "prop drilling."

Prop drilling can make your code more difficult to read and maintain, and can also make it harder to refactor your components later on.

This is where the Context API comes in. With Context API, you can store data at the top level of the component tree and make it available to all other components that need it without passing props.

How the Context API Works

Context API allows data to be passed through a component tree without having to pass props manually at every level. This makes it easier to share data between components.



A diagram illustrating how Context API works

Create a Context Object

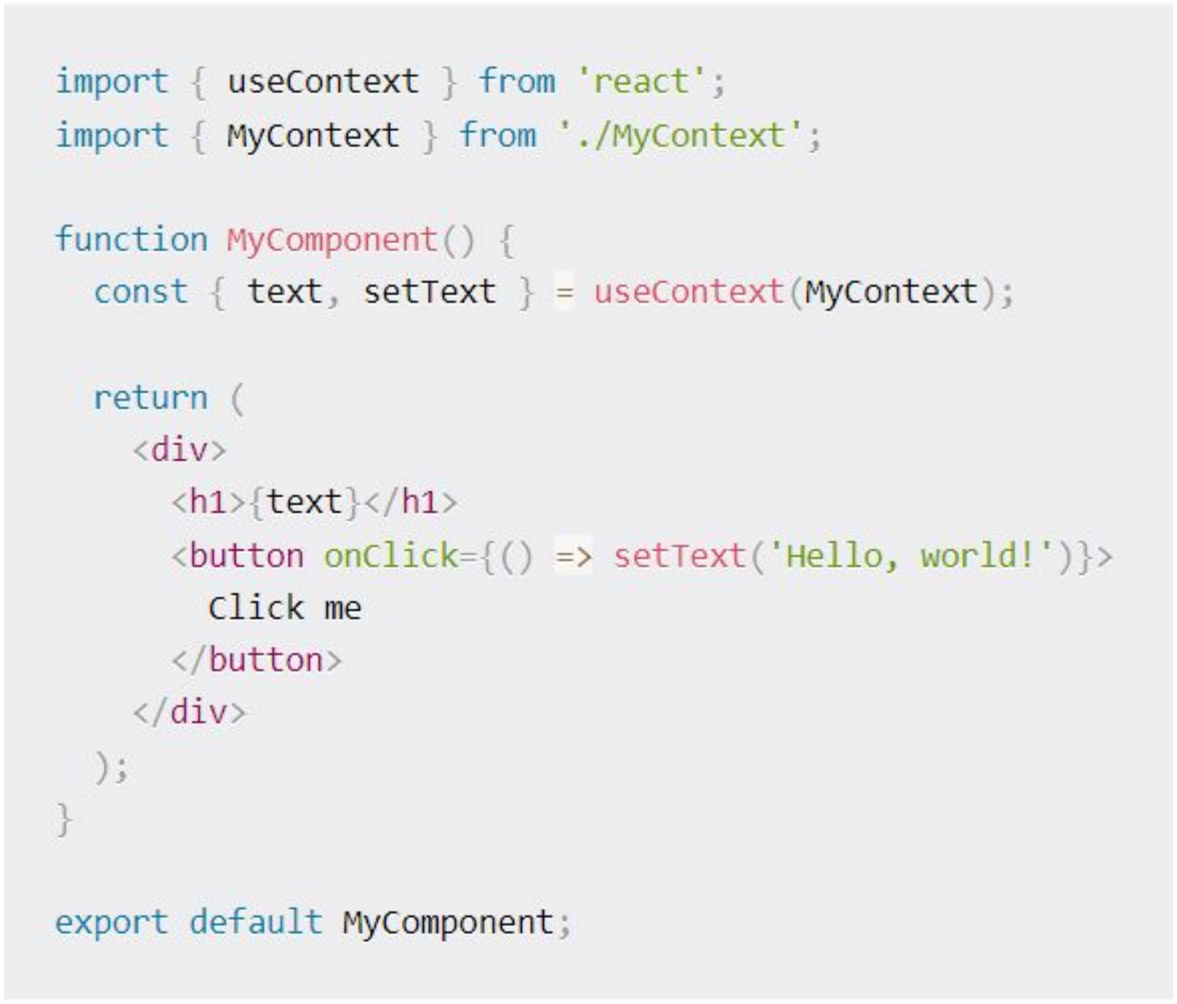
First, you need to create a context object using the createContext function from the 'react' library. This context object will hold the data that you want to share across your application.

Create a new file named MyContext.js in the src folder and add the following code to create a context object:

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

export const MyContext = createContext("");
```

```
// Create a parent component that wraps child components with a Provider

import { useState, React } from "react";
import { MyContext } from "./MyContext";
import MyComponent from "./MyComponent";

function App() {
  const [text, setText] = useState("");

  return (
    <div>
      <MyContext.Provider value={{ text, setText }}>
        <MyComponent />
      </MyContext.Provider>
    </div>
  );
}

export default App;
```

Now that we've created the provider component, we need to consume the context in other components. To do this, we use the "useContext" hook from React.



```
import { useContext } from 'react';
import { MyContext } from './MyContext';

function MyComponent() {
  const { text, setText } = useContext(MyContext);

  return (
    <div>
      <h1>{text}</h1>
      <button onClick={() => setText('Hello, world!')}>
        Click me
      </button>
    </div>
  );
}

export default MyComponent;
```

In this example, we've used the useContext hook to access the "text" and "setText" variables that were defined in the provider component.