**React.js Notes**

Q.What is a wrapper component in React?

Unknown components are surrounded by wrapper components, which give a default structure for displaying the child components. This pattern is excellent for designing user interface (UI) elements like modals, template pages, and information tiles that are used repeatedly throughout a design.

What are the benefits of using a Wrapper Component?

Wrapper components are a way to decouple the logic of your application from its presentation. They allow you to reuse and share the logic of your application without sharing its user interface.

What is the difference between a React component and a React wrapper component?

The difference between a React component and a React wrapper component is that the former includes a render function while the latter wraps an existing DOM element with new functionality and data.

What is the purpose of wrapping components?

Wrapping components is a technique that allows developers to add an extra layer of abstraction on top of the existing system and make it more flexible.

### **What's a wrapper component in react?**

Wrapper components are components that surround unknown components and provide a default structure to display the child components.

A wrapper component may be used to create user interface (UI) components that are used frequently throughout a design, such as modals, template pages, and information tiles.

### Code example of a modal react wrapper component.

export const Modal = (props) => {

const { visible, cancel, children } = props;

const modalRef = useRef(null);

useEffect(() => {

const checkIfClickedOutside = (e) => {

// If the modal is open and the clicked target is not within the modal,

//then close the modal

if (visible && modalRef.current && !modalRef.current.contains(e.target)) {

cancel();

}

};

document.addEventListener("mousedown", checkIfClickedOutside);

// Cleanup the event listener

return () => {

document.removeEventListener("mousedown", checkIfClickedOutside);

};

}, [visible]);

return (

<>

{visible && (

<div className="modal-bg">

<div ref={modalRef} className="modal-container">

<button onClick={cancel}>close modal</button>

{children}

</div>

</div>

)}

</>

);

};

The preceding code demonstrates a react modal component.

The modal component is given the following properties to make it reusable anywhere a modal is needed.

* visible A Boolean value that controls the visibility of the modals.
* cancel A handler function that negates the visibile value.
* children displays whatever you include between the opening and closing tags when invoking the modal wrapper.

The modal component then transforms into a wrapper component, wrapping the children prop and giving it a modal view.

The modal component can be utilized anywhere a modal is required.

function App() {

const [modalVisble, setModalVisble] = useState(false);

const handleOpenModal = () => {

setModalVisble(true);

};

const handleCloseModal = () => {

setModalVisble(false);

};

return (

<div className="App">

<button type="button" onClick={handleOpenModal}>

Open modal

</button>

<Modal visible={modalVisble} cancel={handleCloseModal}>

<h1>Hello World</h1>

<p>I am a modal</p>

</Modal>

</div>

);

}

.modal-bg {

background-color: rgba(0, 0, 0, 0.2);

height: 100vh;

width: 100%;

position: fixed;

top: 0;

left: 0;

display: flex;

justify-content: center;

align-items: center;

}

.modal-container {

background-color: white;

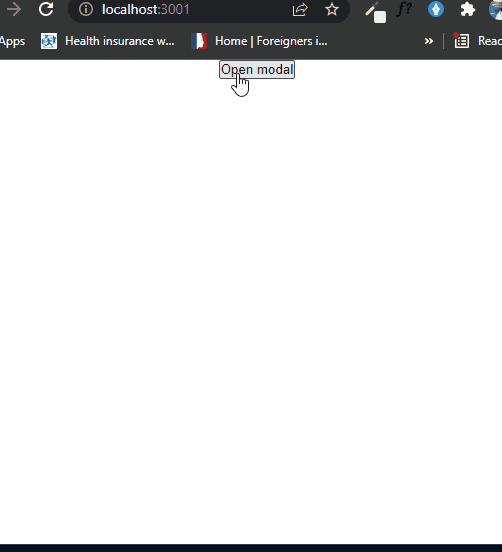
padding: 20px;

width: 70%;

margin: 0 auto;

}

After rendering the page will look like this

GIF

### Conclusion

React wrapper is a must-have approach for every professional react developer.  
In this article we learned how we can create a react wrapper component and its use cases.

**React Fragments**

## What is React Fragment?[​](https://refine.dev/blog/how-react-fragments-is-works/#what-is-react-fragment)

React Fragment is a feature in React that allows you to return multiple elements from a React component by allowing you to group a list of children without adding extra nodes to the DOM.

To return multiple elements from a React component, you'll need to wrap the element in a root element. This approach has not been efficient and may cause issues in some cases. Eg.

*function* TableData () {  
 *return* (  
 <div>  
 <td>Eat</td>  
 <td>Learn</td>  
 <td>Code</td>  
 </div>  
 );  
}  
  
*function* Table () {  
 *return* (  
 <table>  
 <tr>  
 <TableData />  
 </tr>  
 </table>  
 );  
}

The above code will produce the HTML equivalent below.

*function* TableData () {  
 *return* (  
 <div>  
 <td>Eat</td>  
 <td>Learn</td>  
 <td>Code</td>  
 </div>  
 );  
}  
  
*function* Table () {  
 *return* (  
 <table>  
 <tr>  
 <TableData />  
 </tr>  
 </table>  
 );  
}

The above code will produce the HTML equivalent below.

<table>  
 <tr>  
 <div>  
 <td>Eat</td>  
 <td>Learn</td>  
 <td>Code</td>  
 </div>  
 </tr>  
</table>

So as you can see that wrapping the <td> tags in a div element breaks the table parent-child relationship. For things to work as expected, the <td> tags have to be rendered individually without wrapping them in a div element. In scenarios like this, it's better to use React Fragment.

## React Fragment vs Div Element[​](https://refine.dev/blog/how-react-fragments-is-works/#react-fragment-vs-div-element)

In React, "Fragment" and "Div" are used interchangeably. The main difference between the two is that "Fragment" clears out all extra divs from a DOM tree while "Div" adds a div to the DOM tree.

With React Fragments, we can create code that is cleaner and easier to read. It renders components more quickly and uses less memory. Every element is rendered as intended. While Div expands the DOM due to the long nested nodes that occur when there are too many HTML tags on your website.

The div element has more methods and properties, which causes it to consume more memory which can make the page slow load time; the prototype chain is like HTMLDivElement -> HTMLElement -> Element -> Node -> EventTarget, whereas the React fragment has fewer methods with the prototype chain DocumentFragment -> Node -> EventTarget.

Using fragments, you can reuse parts of your application. But, like in the table example we used in the previous section, div makes it challenging to do so. However, there are situations where using div instead of a fragment is necessary.

For instance, utilizing fragments does not allow you to design a component since you must wrap the target elements in a div. Additionally, you must use a div if you are adding keys to the components' elements. In light of this, you can use the two interchangeably depending on what you want your React application to accomplish.

## Problem with using div[​](https://refine.dev/blog/how-react-fragments-is-works/#problem-with-using-div)

Let's look at some of the problems in using div in detail.

* The div element expands the HTML DOM, causing the browser to consume more resources than expected.
* When the DOM is too large, it consumes a lot of memory, causing the pages to load slowly in the browser.
* Debugging and tracing the origin of the extra nodes becomes more difficult as the DOM grows larger and more nested.
* Using div to render components may cause performance issues by clogging your HTML.

## Advantages of Fragment[​](https://refine.dev/blog/how-react-fragments-is-works/#advantages-of-fragment)

React Fragment replaces the <div> element, which can cause issues with invalid HTML, with the following advantages.

* The code readability of React Fragment is higher.
* Because React fragments have a smaller DOM, they render faster and use less memory.
* React Fragment allows React components to be rendered as intended without causing any parent-child relationship issues.
* Fragments allow the return of multiple JSX elements, which addresses the issue of invalid HTML markups within react applications that were caused by the must-have constraint of only one element returning per component.

## Using the key prop with React fragments[​](https://refine.dev/blog/how-react-fragments-is-works/#using-the-key-prop-with-react-fragments)

In some cases, the key prop is required in a React application. In react, the key props are typically used to control component instances. React uses the key prop in scenarios like this to identify which items changed, removed, or added. Using the key props in a React application with fragments will be like the code snippet below.

*function* TableData () {  
 *return* (  
 {data.map(rec=>  
 <React.Fragment key={rec.id}>  
 <td>{rec.hobby}</td>  
 </React.Fragment>  
 )}  
 );  
}

## Using shortcut version[​](https://refine.dev/blog/how-react-fragments-is-works/#using-shortcut-version)

Aside from using React Fragment, React also provides a shorthand notation <></> to wrap multiple elements together that works similarly to React Fragment but with a lower memory load. In a react application, the shorthand notation <></> is implemented as follows.

<*function* TableData () {  
 *return* (  
 <>  
 <td>Eat</td>  
 <td>Learn</td>  
 <td>Code</td>  
 <>  
 );  
}

The above code will produce the expected HTML equivalent below.

<table>  
 <tr>  
 <div>  
 <td>Eat</td>  
 <td>Learn</td>  
 <td>Code</td>  
 </div>  
 </tr>  
</table>

However, there are some drawbacks to this approach. For example, implementing the key props is impossible because the shorthand notation <></> will not work here. After all, it cannot take an attribute.

## Fragment in Action[​](https://refine.dev/blog/how-react-fragments-is-works/#fragment-in-action)

Now let's see how fragments are used in a React application. In the example below, we'll use the React Fragment to render a list of items in a table.

*import* "./App.css";  
*import* React *from* "react";  
  
*const* Table = ({ children, style }) => {  
 *return* <div>{children}</div>;  
};  
  
*const* TableData = () => {  
 *return* (  
 <React.Fragment>  
 <td>John Doe</td>  
 <td>16</td>  
 <td>Developer</td>;  
 </React.Fragment>  
 );  
}  
  
*function* App() {  
 *return* (  
 <Table>  
 <tr>  
 <th>Name</th>  
 <th>Age</th>  
 <th>Occupation</th>  
 </tr>  
 <TableData />  
 </Table>  
 );  
}  
*export* *default* App;

In the above code snippet, we created two components that we to be rendered in our application. In the render method, we used React Fragment instead of wrapping the elements in the TableData components in a div. This way, our table data will be rendered as expected.

## Conclusion[​](https://refine.dev/blog/how-react-fragments-is-works/#conclusion)

Throughout this tutorial, you've learned about React Fragment. We started by knowing what a React Fragment is and when to use it in a React application. Then we went further to demonstrate how it's used in a practical application.

React portals

React 16, released in September 2017, introduced Portals.

A portal is a way to render an element outside of its component hierarchy, in a separate component.

When that event is rendered, events happening on it are managed by the React components hierarchy rather than by the hierarchy set by the DOM position of the element.

Hence the name “portal”: an element sits somewhere in the DOM tree that’s outside of the normal React components tree, but the React component tree that includes it is still in charge.

React offers an easy API to do this, ReactDOM.createPortal(), which accepts 2 arguments. The first is the element to render, the second is the DOM element where to render it.

A classic use case for this is modal windows.

A modal to render at full screen must live outside of the element, so it can be properly styled using CSS.

So if a modal is defined as a component:

class Modal extends React.Component {

constructor(props) {

super(props)

this.el = document.createElement('div')

}

componentDidMount() {

document.getElementById('modal').appendChild(this.el)

}

componentWillUnmount() {

document.getElementById('modal').removeChild(this.el)

}

render() {

return ReactDOM.createPortal(

this.props.children,

this.el

)

}

}

We can have an App component render it, and all the events happening in the Modal component will be handled by App even though technically the modal is rendered in a different DOM tree:

class App extends React.Component {

constructor(props) {

super(props)

this.state = {showModal: false}

this.handleShow = this.handleShow.bind(this)

this.handleHide = this.handleHide.bind(this)

}

handleShow() {

this.setState({showModal: true})

}

handleHide() {

this.setState({showModal: false})

}

render() {

const modal = this.state.showModal ? (

<Modal>

<div>

The modal <button onClick={this.handleHide}>Hide</button>

</div>

</Modal>

) : ''

return (

<div>

The app <button onClick={this.handleShow}>Show modal</button>

{modal}

</div>

)

}

}

ReactDOM.render(<App />, document.getElementById('app'))

**React Refs(References)**

In React, a ref is a way to access the properties of a DOM element. Refs are created using the **React.createRef()** method, and they can be attached to React elements via the **ref** attribute. Once a ref is attached to an element, the ref can be used to get the value of the element's property, such as its **value**, **checked**, or **innerHTML**.

Refs are primarily used for two purposes:

1. To access the value of an input element, such as a text input or a checkbox.
2. To access the DOM API, such as to measure the size of an element or to focus an input element.

Here is an example of using a ref to get the value of an input element:

*import React from 'react';*

*class MyForm extends React.Component {*

*inputRef = React.createRef();*

*handleSubmit = (event) => {*

*event.preventDefault();*

*console.log(this.inputRef.current.value);*

*}*

*render() {*

*return (*

*<form onSubmit={this.handleSubmit}>*

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

*<button type="submit">Submit</button>*

*</form>*

*);*

*}*

*}*

In this example, the **inputRef** ref is created using **React.createRef()** and attached to the **<input>** element via the **ref** attribute. When the form is submitted, the **handleSubmit** method is called, and it logs the value of the input element to the console using **this.inputRef.current.value**.

Refs can also be used to access the DOM API, such as to focus an input element or measure the size of an element. Here is an example of using a ref to focus an input element:

In this example, the **inputRef** ref is created and attached to the **<input>** element as before. The **componentDidMount** lifecycle method is then used to focus the input element using **this.inputRef.current.focus()**. This will cause the input element to receive focus as soon as the component is mounted.

*import React from 'react';*

*class MyForm extends React.Component {*

*inputRef = React.createRef();*

*componentDidMount() {*

*this.inputRef.current.focus();*

*}*

*render() {*

*return (*

*<form>*

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

*</form>*

*);*

*}*

*}*

## When to Use Refs

Refs can be used in the following cases:

* When we need DOM measurements such as managing focus, text selection, or media playback.
* It is used in triggering imperative animations.
* When integrating with third-party DOM libraries.
* It can also use as in callbacks.

## When to not use Refs

* Its use should be avoided for anything that can be done **declaratively**. For example, instead of using **open()** and **close()** methods on a Dialog component, you need to pass an **isOpen** prop to it.
* You should have to avoid overuse of the Refs.

Note:-> useState is a controlled state and refs is uncontrolled state. So, refs is mostly should be avoided to use and can only used in taking input values and to focus at the input values.