**Understanding Ref Hooks**

**Refs and the DOM**

1. Refs provide a way to access DOM nodes or Elements in react create while rendering
2. In a typical dataflow, props are the only way that **parent component** interact with their **child component.**
3. So, to modify a child component we need to re-render with a new prop.
4. The child component to be modified we can create an instance of a react component, or it a DOM element.

**When to Use Ref**

Few of the cases are:

1. Managing the focus, text selection
2. Media playback
3. Interacting with third party DOM libraries

**Don’t overuse Ref**

What I meant is instead of using open() and close() methods on a dialogue component, we can pass an isOpen prop to it.

**Creating Refs**

1. Refs are created using React.createRef() and are attached to the element in React
2. Refs are commonly assigned to an instance property when a component is constructed so that they can be use for reference between different components.

Example of callback ref

///Cretaing fres / callback Ref example

const { render } = require("react-dom");

Class MyComponent extends React.Coponent {

    constructor(props) {

        super(props);

        this.myRef = React.createRef();

    }

    render () {

        return <div ref={this.myRef}/>;

    }

}

**Accessing Refs**

When a ref is passed to an elemnt in render, a reference to the node becomes accessible at the current attribute of the ref.

Const node = this.myRef.current;

The value of the ref differs depending on the type of the node:

When the ref attribute is used on an element(HTML), the created ref in the constructor with React.createRef() receives the element as its current property.

And, when the ref attribute is used on a custom class component, the ref receives the mounted instance of the component.

//Example1

//Below example uses a ref to store a reference to DOM node:

class CustomTextInput extends React.Component {

    constructor(props) {

        super(props);

        //ref is created over a DOM element

        this.textInput = React.createRef();

        this.focusTextInput = this.focusTextInput.bind(this);

    }

    focusTextInput() {

        //We are accessing the "current" to get the DOM node

        this.textInput.current.focus();

    }

    render() {

        return (

            <div>

                <input

                type="text"

                ref={this.textInput}/>

                <input

                type="button"

                value="Focus the text inputted"

                onClick={this.focusTextInput}/>

            </div>

        )

    }

}

Refs updates are happened before componentDidMount or componentDidUpdate lifecycle methods.

Example2: Adding ref to a class component

//Example2: Adding Ref to a class Component

//If we want to wrap the Custom TextInput explained in the above example it being clicked immidiately after mounting, we could use a

//ref to get access to the custom input and call its focusTextInput method manually.

class AutoFocusTextInput extends React.Component {

    constructor(props) {

        super(props);

        this.textInput = React.createRef();

    }

    componentDidMount() {

        this.textInput.current.focusTextInput();

    }

    render () {

        return (

            <CustomTextInput ref={this.textInput}/>

        );

    }

}

Example3: Refs in function component

//Example3: Refs and Function components

function MyFunctionCompoent() {

    return <input/>;

}

class Parent extends React.Component {

    constructor(props) {

        super (props);

        this.textInput = React.createRef();

    }

    render() {

        ///this wont work

        return(

            <MyFunctionCompoent ref={this.textInput} />

        );

    }

}

//inside a function

function CustomTextInput(props) {

    const textInput = useRef(null);

    function handleClick() {

        textInput.current.focus();

    }

    return (

        <div>

            <input

                type="text"

                ref={textInput}/>

                <input

                type="button"

                value="Focus the text inputted"

                onClick={handleClick}/>

        </div>

    )

}

Exposing DOM Refs to Parent Components

In rare cases, you might want to have access to a child’s DOM node from a parent component.

This is generally not recommended because it breaks component encapsulation, but it can occasionally be useful for triggering focus or measuring the size

or position of a child DOM node.

While you could add a ref to the child component, this is not an ideal solution, as you would only get a component instance rather than a DOM node.

Additionally, this wouldn’t work with function components.

If you use React 16.3 or higher, it is recommend to use ref forwarding for these cases. Ref forwarding lets components opt into exposing any child

component’s ref as their own.

You can find a detailed example of how to expose a child’s DOM node to a parent component in the ref forwarding documentation.

Explained below in this document.

If you use React 16.2 or lower, or if you need more flexibility than provided by ref forwarding,

you can use this alternative approach and explicitly pass a ref as a differently named prop.

When possible, it is advised against exposing DOM nodes, but it can be a useful escape hatch. Note that this approach requires you to add some

code to the child component.

If you have absolutely no control over the child component implementation, your last option is to use findDOMNode(),

but it is discouraged and deprecated in StrictMode.

**Callback Refs**

React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.

Instead of passing a ref attribute created by createRef(), you pass a function. The function receives the React component instance or HTML DOM element

as its argument, which can be stored and accessed elsewhere.

The example below implements a common pattern: using the ref callback to store a reference to a DOM node in an instance property.

//The example below implements a common pattern: using the ref callback to store a reference to a DOM node in an instance property.

class CustomTextInput extends React.Component {

  constructor(props) {

    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {

      this.textInput = element;

    };

    this.focusTextInput = () => {

      // Focus the text input using the raw DOM API

      if (this.textInput) this.textInput.focus();

    };

  }

  componentDidMount() {

    // autofocus the input on mount

    this.focusTextInput();

  }

  render() {

    // Use the `ref` callback to store a reference to the text input DOM

    // element in an instance field (for example, this.textInput).

    return (

      <div>

        <input

          type="text"

          ref={this.setTextInputRef}

        />

        <input

          type="button"

          value="Focus the text input"

          onClick={this.focusTextInput}

        />

      </div>

    );

  }

}

React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts.

Refs are guaranteed to be up-to-date before componentDidMount or componentDidUpdate fires.

You can pass callback refs between components like you can with object refs that were created with React.createRef().

function CustomTextInput(props) {

    return (

      <div>

        <input ref={props.inputRef} />

      </div>

    );

  }

  class Parent extends React.Component {

    render() {

      return (

        <CustomTextInput

          inputRef={el => this.inputElement = el}

        />

      );

    }

  }

In the example above, Parent passes its ref callback as an inputRef prop to the CustomTextInput, and the CustomTextInput passes the same function as a

special ref attribute to the <input>.

As a result, this.inputElement in Parent will be set to the DOM node corresponding to the <input> element in the CustomTextInput.

//useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue).

//The returned object will persist for the full lifetime of the component.

//A common use case is to access a child imperatively:

function TextInputWithFocusButton() {

  const inputEl = useRef(null);

  const onButtonClick = () => {

    // `current` points to the mounted text input element

    inputEl.current.focus();

  };

  return (

    <>

      <input ref={inputEl} type="text" />

      <button onClick={onButtonClick}>Focus the input</button>

    </>

  );

}

Essentially, useRef is like a “box” that can hold a mutable value in its .current property.

You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with <div ref={myRef} />,

React will set its .current property to the corresponding DOM node whenever that node changes.

However, useRef() is useful for more than the ref attribute. It’s handy for keeping any mutable value around similar to how you’d use instance

fields in classes.

This works because useRef() creates a plain JavaScript object. The only difference between useRef() and creating a {current: ...} object yourself

is that useRef will give you the same ref object on every render.

Keep in mind that useRef doesn’t notify you when its content changes. Mutating the .current property doesn’t cause a re-render.

If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a callback ref instead.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Forwarding Refs**

Ref forwarding is a technique for automatically passing a ref through a component to one of its children.

This is typically not necessary for most components in the application.

However, it can be useful for some kinds of components, especially in reusable component libraries.

The most common scenarios are described below.

Forwarding refs to DOM components

Consider a Button component that renders the native button DOM element:

//Forwarding ref in DOM

function FancyButton(props) {

    return (

      <button className="FancyButton">

        {props.children}

      </button>

    );

  }

React components hide their implementation details, including their rendered output.

Other components using FancyButton usually will not need to obtain a ref to the inner button DOM element.

This is good because it prevents components from relying on each other’s DOM structure too much.

Although such encapsulation is desirable for application-level components like FeedStory or Comment, it can be inconvenient for highly reusable “leaf”

components like FancyButton or MyTextInput.

These components tend to be used throughout the application in a similar manner as a regular DOM button and input, and accessing their DOM nodes

may be unavoidable for managing focus, selection, or animations.

Ref forwarding is an opt-in feature that lets some components take a ref they receive, and pass it further down (in other words, “forward” it) to a child.

In the example below, FancyButton uses React.forwardRef to obtain the ref passed to it, and then forward it to the DOM button that it renders:

 const FancyButton = React.forwardRef((props, ref) => (

    <button ref={ref} className="FancyButton">

      {props.children}

    </button>

  ));

Step-by-step explanation of what happens in the above example:

We create a React ref by calling React.createRef and assign it to a ref variable.

We pass our ref down to <FancyButton ref={ref}> by specifying it as a JSX attribute.

React passes the ref to the (props, ref) => ... function inside forwardRef as a second argument.

We forward this ref argument down to <button ref={ref}> by specifying it as a JSX attribute.

When the ref is attached, ref.current will point to the <button> DOM node.

Note

The second ref argument only exists when you define a component with React.forwardRef call.

Regular function or class components don’t receive the ref argument, and ref is not available in props either.

Ref forwarding is not limited to DOM components. You can forward refs to class component instances, too.

Note for component library maintainers

When you start using forwardRef in a component library, you should treat it as a breaking change and release a new major version of your library.

This is because your library likely has an observably different behavior (such as what refs get assigned to, and what types are exported),

and this can break apps and other libraries that depend on the old behavior.

Conditionally applying React.forwardRef when it exists is also not recommended for the same reasons: it changes how your library behaves and can break

your users’ apps when they upgrade React itself.

Forwarding refs in higher-order components

This technique can also be particularly useful with higher-order components (also known as HOCs).

Let’s start with an example HOC that logs component props to the console:

//Forwarding ref in High Order component

  function logProps(WrappedComponent) {

    class LogProps extends React.Component {

      componentDidUpdate(prevProps) {

        console.log('old props:', prevProps);

        console.log('new props:', this.props);

      }

      render() {

        return <WrappedComponent {...this.props} />;

      }

    }

    return LogProps;

  }

// Rather than exporting FancyButton, we export LogProps.

// It will render a FancyButton though.

export default logProps(FancyButton);

There is one caveat to the above example: refs will not get passed through. That’s because ref is not a prop. Like key, it’s handled differently by React.

If you add a ref to a HOC, the ref will refer to the outermost container component, not the wrapped component.

This means that refs intended for our FancyButton component will actually be attached to the LogProps component:

  import FancyButton from './FancyButton';

const ref = React.createRef();

// The FancyButton component we imported is the LogProps HOC.

// Even though the rendered output will be the same,

// Our ref will point to LogProps instead of the inner FancyButton component!

// This means we can't call e.g. ref.current.focus()

<FancyButton

  label="Click Me"

  handleClick={handleClick}

  ref={ref}

/>;

Fortunately, we can explicitly forward refs to the inner FancyButton component using the React.forwardRef API.

React.forwardRef accepts a render function that receives props and ref parameters and returns a React node. For example:

function logProps(Component) {

    class LogProps extends React.Component {

      componentDidUpdate(prevProps) {

        console.log('old props:', prevProps);

        console.log('new props:', this.props);

      }

      render() {

        const {forwardedRef, ...rest} = this.props;

        // Assign the custom prop "forwardedRef" as a ref

        return <Component ref={forwardedRef} {...rest} />;

      }

    }

    // Note the second param "ref" provided by React.forwardRef.

    // We can pass it along to LogProps as a regular prop, e.g. "forwardedRef"

    // And it can then be attached to the Component.

    return React.forwardRef((props, ref) => {

      return <LogProps {...props} forwardedRef={ref} />;

    });

  }

Displaying a custom name in DevTools

React.forwardRef accepts a render function. React DevTools uses this function to determine what to display for the ref forwarding component.

For example, the following component will appear as ”ForwardRef” in the DevTools:

const WrappedComponent = React.forwardRef((props, ref) => {

return <LogProps {...props} forwardedRef={ref} />;

});

If you name the render function, DevTools will also include its name (e.g. ”ForwardRef(myFunction)”):

const WrappedComponent = React.forwardRef(

function myFunction(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

);

You can even set the function’s displayName property to include the component you’re wrapping:

function logProps(Component) {

class LogProps extends React.Component {

// ...

}

function forwardRef(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

// Give this component a more helpful display name in DevTools.

// e.g. "ForwardRef(logProps(MyComponent))"

const name = Component.displayName || Component.name;

forwardRef.displayName = `logProps(${name})`;

return React.forwardRef(forwardRef);

}