React Refs

Refs is the shorthand used for **references** in React. It is similar to **keys** in React. It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements. It provides a way to access React DOM nodes or React elements and how to interact with it. It is used when we want to change the value of a child component, without making the use of props.

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.

How to create Refs

In React, Refs can be created by using **React.createRef()**. It can be assigned to React elements via the **ref** attribute. It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.callRef = React.createRef();
  }
  render() {
    return <div ref={this.callRef} />;
  }
}
```

How to access Refs

In React, when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.

```
const node = this.callRef.current;
```

Refs current Properties

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

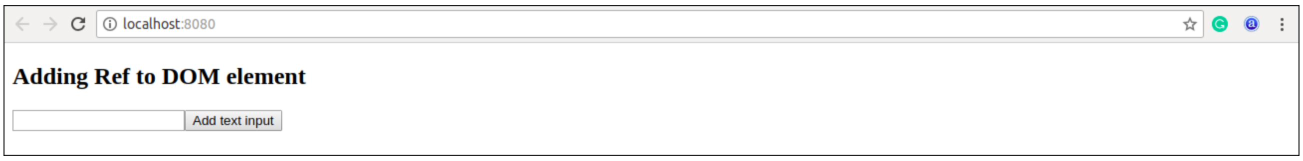
- When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
- If the ref attribute is used on a custom class component, then ref object receives the **mounted** instance of the component as its current property.
- The ref attribute cannot be used on **function components** because they don't have instances.

Add Ref to DOM elements

In the below example, we are adding a ref to store the reference to a DOM node or element.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.callRef = React.createRef();
    this.addingRefInput = this.addingRefInput.bind(this);
  }

  addingRefInput() {
    this.callRef.current.focus();
  }

  render() {
    return (
      <div>
        <h2>Adding Ref to DOM element</h2>
        <input
          type="text"
          ref={this.callRef} />
        <input
          type="button"
          value="Add text input"
          onClick={this.addingRefInput}
        />
      </div>
    );
  }
}
export default App;
```

Output


localhost:8080

Adding Ref to DOM element

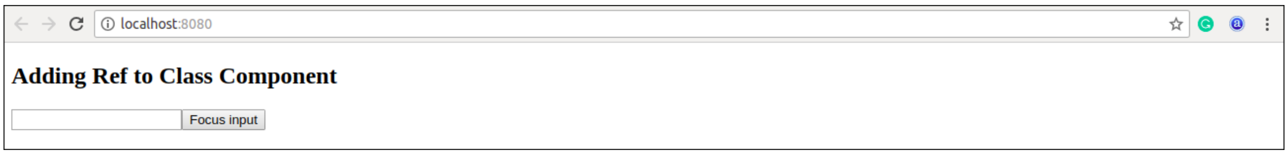
Add Ref to Class components

In the below example, we are adding a ref to store the reference to a class component.

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';

function CustomInput(props) {
  let callRefInput = React.createRef();

  function handleClick() {
    callRefInput.current.focus();
  }

  return (
    <div>
      <h2>Adding Ref to Class Component</h2>
      <input
        type="text"
        ref={callRefInput} />
      <input
        type="button"
        value="Focus input"
        onClick={handleClick}
      />
    </div>
  );
}
class App extends React.Component {
  constructor(props) {
    super(props);
    this.callRefInput = React.createRef();
  }

  focusRefInput() {
    this.callRefInput.current.focus();
  }

  render() {
    return (
      <CustomInput ref={this.callRefInput} />
    );
  }
}
export default App;
```

Output


localhost:8080

Adding Ref to Class Component

Callback refs

In react, there is another way to use refs that is called "**callback refs**" and it gives more control when the refs are **set** and **unset**. Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component. It looks like the below code.

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

The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere. It can be accessed as below:

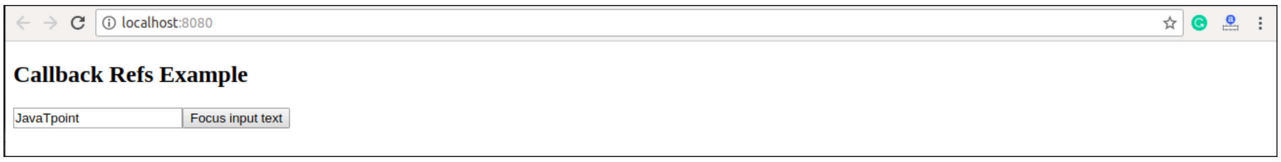
```
this.callRefInput.value
```

The example below helps to understand the working of callback refs.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
    constructor(props) {
    super(props);

    this.callRefInput = null;

    this.setInputRef = element => {
      this.callRefInput = element;
    };

    this.focusRefInput = () => {
      //Focus the input using the raw DOM API
      if (this.callRefInput) this.callRefInput.focus();
    };
  }

  componentDidMount() {
    //autofocus of the input on mount
    this.focusRefInput();
  }

  render() {
    return (
      <div>
    <h2>Callback Refs Example</h2>
        <input
          type="text"
          ref={this.setInputRef}
        />
        <input
          type="button"
          value="Focus input text"
          onClick={this.focusRefInput}
        />
      </div>
    );
  }
}
export default App;
```

In the above example, React will call the "ref" callback to store the reference to the input DOM element when the component **mounts**, and when the component **unmounts**, call it with **null**. Refs are always **up-to-date** before the **componentDidMount** or **componentDidUpdate** fires. The callback refs pass between components is the same as you can work with object refs, which is created with React.createRef().

Output



localhost:8080

Callback Refs Example

JavaTpoint | Focus input text

Forwarding Ref from one component to another component

Ref forwarding is a technique that is used for passing a **ref** through a component to one of its child components. It can be performed by making use of the **React.forwardRef()** method. This technique is particularly useful with **higher-order components** and specially used in reusable component libraries. The most common example is given below.

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';

const TextInput = React.forwardRef((props, ref) => (
  <input type="text" placeholder="Hello World" ref={ref} />
));

const inputRef = React.createRef();

class CustomTextInput extends React.Component {
  handleSubmit = e => {
    e.preventDefault();
    console.log(inputRef.current.value);
  };
  render() {
    return (
      <div>
        <form onSubmit={e => this.handleSubmit(e)}>
          <TextInput ref={inputRef} />
          <button>Submit</button>
        </form>
      </div>
    );
  }
}
export default App;
```

In the above example, there is a component **TextInput** that has a child as an input field. Now, to pass or forward the **ref** down to the input, first, create a ref and then pass your ref down to **<TextInput ref={inputRef}>**. After that, React forwards the ref to the **forwardRef** function as a second argument. Next, we forward this ref argument down to **<input ref={ref}>**. Now, the value of the DOM node can be accessed at **inputRef.current**.

React with useRef()

It is introduced in **React 16.7** and above version. It helps to get access the DOM node or element, and then we can interact with that DOM node or element such as focussing the input element or accessing the input element value. It returns the ref object whose **.current** property initialized to the passed argument. The returned object persist for the lifetime of the component.

Syntax

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

Example

In the below code, **useRef** is a function that gets assigned to a variable, **inputRef**, and then attached to an attribute called ref inside the HTML element in which you want to reference.

```
function useRefExample() {
  const inputRef= useRef(null);
  const onButtonClick = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={onButtonClick}>Submit</button>
    </>
  );
}
```

React Fragments

In React, whenever you want to render something on the screen, you need to use a render method inside the component. This render method can return **single** elements or **multiple** elements. The render method will only render a single root node inside it at a time. However, if you want to return multiple elements, the render method will require a '**div**' tag and put the entire content or elements inside it. This extra node to the DOM sometimes results in the wrong formatting of your HTML output and also not loved by the many developers.

Example

```
// Rendering with div tag
class App extends React.Component {
    render() {
     return (
        //Extraneous div element
        <div>
          <h2> Hello World! </h2>
          <p> Welcome to the JavaTpoint. </p>
        </div>
     );
    }
}
```

To solve this problem, React introduced **Fragments** from the **16.2** and above version. Fragments allow you to group a list of children without adding extra nodes to the DOM.

Syntax

```
<React.Fragment>
      <h2> child1 </h2>
    <p> child2 </p>
      .. ..... .... ...
</React.Fragment>
```

Example

```
// Rendering with fragments tag
class App extends React.Component {
   render() {
    return (
      <React.Fragment>
           <h2> Hello World! </h2>
       <p> Welcome to the JavaTpoint. </p>
        </React.Fragment>
     );
   }
}
```

Why we use Fragments?

The main reason to use Fragments tag is:

1. It makes the execution of code faster as compared to the div tag.
2. It takes less memory.

Fragments Short Syntax

There is also another shorthand exists for declaring fragments for the above method. It looks like **empty** tag in which we can use of '<>' and '' instead of the '**React.Fragment**'.

Example

```
//Rendering with short syntax
class Columns extends React.Component {
  render() {
    return (
      <>
        <h2> Hello World! </h2>
        <p> Welcome to the JavaTpoint </p>
      </>
    );
  }
}
```

Keyed Fragments

The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list. If you need to provide keys, you have to declare the fragments with the explicit <**React.Fragment**> syntax.

Note: Key is the only attributes that can be passed with the Fragments.

Example

```
Function  = (props) {
  return (
    <Fragment>
      {props.items.data.map(item => (
        // Without the 'key', React will give a key warning
        <React.Fragment key={item.id}>
          <h2>{item.name}</h2>
          <p>{item.url}</p>
          <p>{item.description}</p>
        </React.Fragment>
      ))}
    </Fragment>
  )
}
```