Refs and the DOM

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

Don’t Overuse Refs

Your first inclination may be to use refs to “make things happen” in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to “own” that state is at a higher level in the hierarchy. See the Lifting State Up guide for examples of this.

> Note
>
> The examples below have been updated to use the `React.createRef()` API introduced in React 16.3. If you are using an earlier release of React, we recommend using callback refs instead.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

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

Accessing Refs

When a ref is passed to an element 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 HTML element, the `ref` created in the constructor with `React.createRef()` receives the underlying DOM element as its `current` property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its `current`.
- **You may not use the `ref` attribute on functional components** because they don't have instances.

The examples below demonstrate the differences.

Adding a Ref to a DOM Element

This code uses a `ref` to store a reference to a DOM node:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React will assign the `current` property with the DOM element when the component mounts, and assign it back to `null` when it unmounts. `ref` updates happen before `componentDidMount` or `componentDidUpdate` lifecycle hooks.

Adding a Ref to a Class Component

If we wanted to wrap the `CustomTextInput` above to simulate it being clicked immediately 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} />
    );
  }
}
```

Note that this only works if `CustomTextInput` is declared as a class:

```
class CustomTextInput extends React.Component {
  // ...
}
```

Refs and Functional Components

You may not use the `ref` attribute on functional components because they don't have instances:

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // This will *not* work!
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

You should convert the component to a class if you need a ref to it, just like you do when you need lifecycle methods or state.

You can, however, **use the `ref` attribute inside a functional component** as long as you refer to a DOM element or a class component:

```
function CustomTextInput(props) {
  // textInput must be declared here so the ref can refer to it
  let textInput = React.createRef();

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

  return (
    <div>
      <input
        type="text"
        ref={textInput} />
      <input
        type="button"
        value="Focus the text input"
        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 functional components.

If you use React 16.3 or higher, we 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.

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, we advise 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.

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.

```
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`.

Legacy API: String Refs

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `"textInput"`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have some issues, are considered legacy, and **are likely to be removed in one of the future releases**.

> Note
>
> If you're currently using `this.refs.textInput` to access refs, we recommend using either the callback pattern or the `createRef` API instead.

Caveats with callback refs

If the `ref` callback is defined as an inline function, it will get called twice during updates, first with `null` and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one. You can avoid this by defining the `ref` callback as a bound method on the class, but note that it shouldn't matter in most cases.