Forwarding Refs

These docs are old and won't be updated. Go to react.dev for the new React docs.
These new documentation pages teach modern React and include live examples:

- Manipulating the DOM with Refs
- `forwardRef`

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 `FancyButton` component that renders the native `button` DOM element:

```
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>
));

// You can now get a ref directly to the DOM button:
```

```
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

This way, components using `FancyButton` can get a ref to the underlying `button` DOM node and access it if necessary—just like if they used a DOM `button` directly.

Here is a step-by-step explanation of what happens in the above example:

1. We create a React ref by calling `React.createRef` and assign it to a `ref` variable.
2. We pass our `ref` down to `<FancyButton ref={ref}>` by specifying it as a JSX attribute.
3. React passes the `ref` to the `(props, ref) => ...` function inside `forwardRef` as a second argument.
4. We forward this `ref` argument down to `<button ref={ref}>` by specifying it as a JSX attribute.
5. 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:

```
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;
}
```

The "logProps" HOC passes all `props` through to the component it wraps, so the rendered output will be the same.

For example, we can use this HOC to log all props that get passed to our “fancy button” component:

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}

// 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);
}
```

Is this page useful?  Edit this page