Controlled / Uncontrolled React Components

Solomon Hawk, Senior Developer

#CODE

Posted on July 31, 2017

SHARE SHARE TWEET

Ever wondered how to author your own controlled or uncontrolled components?

Some Background

If you're new to React application development, you might be asking yourself, "What are controlled and uncontrolled components, anyway?" I suggest taking a look at the docs linked above for a little extra context.

The need for controlled and uncontrolled components in React apps arises from the behavior of certain DOM elements such as `<input>`, `<textarea>`, and `<select>` that by default maintain state (user input) within the DOM layer. Controlled components instead keep that state inside of React either in the component rendering the input, a parent component somewhere in the tree, or a flux store.

However this pattern can be extended to cover certain use cases that are unrelated to DOM state. For example, in a recent application I needed to create a nest-able `Collapsible` component that supported two modes of operation: in some cases it needed to be controlled externally (expanded through user interaction with other areas of the app) and in other cases it could simply manage it's own state.

Inputs in React

For `input`s in React, it works like this.

1.

To create an uncontrolled `input`: set a `defaultValue` prop. In this case the React component will manage the value of its underlying DOM node within local component state.

Implementation details aside, you can think of this as calls to `setState()` within the component to update `state.value` which is assigned to the DOM input.

2.
To create a controlled `input`: set the `value` and `onChange()` props. In this case, React will always assign the `value` prop as the input's value whenever the `value` prop changes. When a user changes the input's value, the `onChange()` callback will be called which must eventually result in a new `value` prop being sent to the input. Consequently, if `onChange()` isn't wired up correctly, the input is effectively read-only; a user cannot change the value of the input because whenever the input is rendered it's value is set to the `value` prop.

The General Pattern

Fortunately it's trivial to author a component with this behavior. The key is to create a component interface that accepts one of two possible configurations of properties.

1.
To create a controlled component, define the property you want to control as `defaultX`. When a component is instantiated and is given a `defaultX` prop, it will begin with the value of that property and will manage its own state over the lifetime of the component (making calls to `setState()` in response to user interaction). This covers use case 1: the component does not need to be externally controlled and state can be local to the component.

2.
To create an uncontrolled component, define the property you want to control as `x`. When a component is instantiated and is given an `x` prop and a callback to change `x`, (e.g. `toggleX()`, if `x` is a boolean) it will begin with the value of that prop. When a user interacts with the component, instead of a `setState()` call within, the component must call the callback `toggleX()` to request that state is externally updated. After that update propagates, the containing component should end up re-rendering and sending a new `x` property to the controlled component.

The Collapsible Interface

For the `Collapsible` implementation, I was only dealing with a boolean property so I chose to use `collapsed` / `defaultCollapsed` and `toggleCollapsed()` for my component interface.

1.
When given a `defaultCollapsed` prop, the Collapsible will begin in the state declared by the prop but will manage it's own state over the lifetime of the component. Clicking on the child `button` will trigger a `setState()` that updates the internal component state.

2.
When given a `collapsed` boolean prop and a `toggleCollapsed()` callback prop, the Collapsible will similarly begin in the state declared by `collapsed` but, when clicked, will only call the `toggleCollapsed()` callback. The expectation is that `toggleCollapsed()` will update state in an ancestor component which will cause the Collapsible to be re-rendered

with a new `collapsed` property after the callback modifies state elsewhere in the application.

Implementation

There is a dead-simple pattern within the component implementation that makes this work. The general idea is:

1.

When the component is instantiated, set its state to the value of `x` that was passed in or the default value for `x`. In the case of the `Collapsible`, the default value of `defaultCollapsed` is `false`.

2.

When rendering, if the `x` prop is defined, then respect it (controlled), otherwise use the local component value in `this.state` (uncontrolled). This means that in `Collapsible`'s `render` method I determine the collapsed state as such:

```
let collapsed = this.props.hasOwnProperty('collapsed') ? this.props.collapsed : this.state.collapsed
```

With destructuring and default values, this becomes satisfyingly elegant:

```
// covers selecting the state for both the controlled and uncontrolled use cases
const {
  collapsed = this.state.collapsed,
  toggleCollapsed
} = this.props
```

The above says, "give me a binding called `collapsed` whose value is `this.props.collapsed` but, if that value is `undefined`, use `this.state.collapsed` instead".

Wrapping Up

I hope you can see how simple and potentially useful it is to support both controlled and uncontrolled behaviors in your own components. I hope you have a clear understanding of *why* you might need to build components in this way and hopefully also *how*. Below I've included a the full source of `Collapsible` in case you're curious - it's pretty short.

```
/**
 * The Collapsible component is a higher order component that wraps a given
 * component with collapsible behavior. The wrapped component is responsible
 * for determining what to render based on the `collapsed` prop that will be
 * sent to it.
 */
import invariant from 'invariant'
import { createElement, Component } from 'react'
import getDisplayName from 'recompose/getDisplayName'
import hoistStatics from 'hoist-non-react-statics'
import PropTypes from 'prop-types'

export default function collapsible(WrappedComponent) {
  invariant(
    typeof WrappedComponent == 'function',
    `You must pass a component to the function returned by ` +
    `collapsible. Instead received ${JSON.stringify(WrappedComponent)}`
  )

  const wrappedComponentName = getDisplayName(WrappedComponent)
  const displayName = `Collapsible(${wrappedComponentName})`

  class Collapsible extends Component {

    static displayName = displayName
    static WrappedComponent = WrappedComponent

    static propTypes = {
      onToggle: PropTypes.func,
      collapsed: PropTypes.bool,
      defaultCollapsed: PropTypes.bool
    }

    static defaultProps = {
      onToggle: () => {},
      collapsed: undefined,
      defaultCollapsed: true
    }

    constructor(props, context) {
      super(props, context)

      this.state = {
        collapsed: props.defaultCollapsed
      }
    }

    render() {
      const {
        collapsed = this.state.collapsed, // the magic
        defaultCollapsed,
        ...props
      } = this.props

      return createElement(WrappedComponent, {
        ...props,
        collapsed,
        toggleCollapsed: this.toggleCollapsed
      })
    }

    toggleCollapsed = () => {
      this.setState(({ collapsed }) => ({ collapsed: !collapsed }))
      this.props.onToggle()
    }
  }

  return hoistStatics(Collapsible, WrappedComponent)
}
```

We do a lot of work with React so we created Microcosm: a batteries-included flux. Read more from Nate and David...

Solomon is a developer in our Durham, NC, office. He focuses on JavaScript development for clients such as ID.me. He loves distilling complex challenges into simple, elegant solutions.

More articles by Solomon

Related Articles

ARTICLE

Introducing Microcosm: Our Data Layer For React

Nate Hunzaker

ARTICLE

Using Microcosm Presenters to Manage Complex Features

David Eisinger

ARTICLE

Embeddable Web Applications with Shadow DOM

Nick Telsan

Have an unsolvable problem or audacious idea?

Let’s get to work

Contact Us →

hello@viget.com

703.891.0670

Sign Up For Our Newsletter

A curated periodical featuring thoughts, opinions, and tools for building a better digital world.

Check it out →

viget

Washington DC Metro | Durham, NC | Boulder, CO | Chattanooga, TN