Published in Level Up Coding

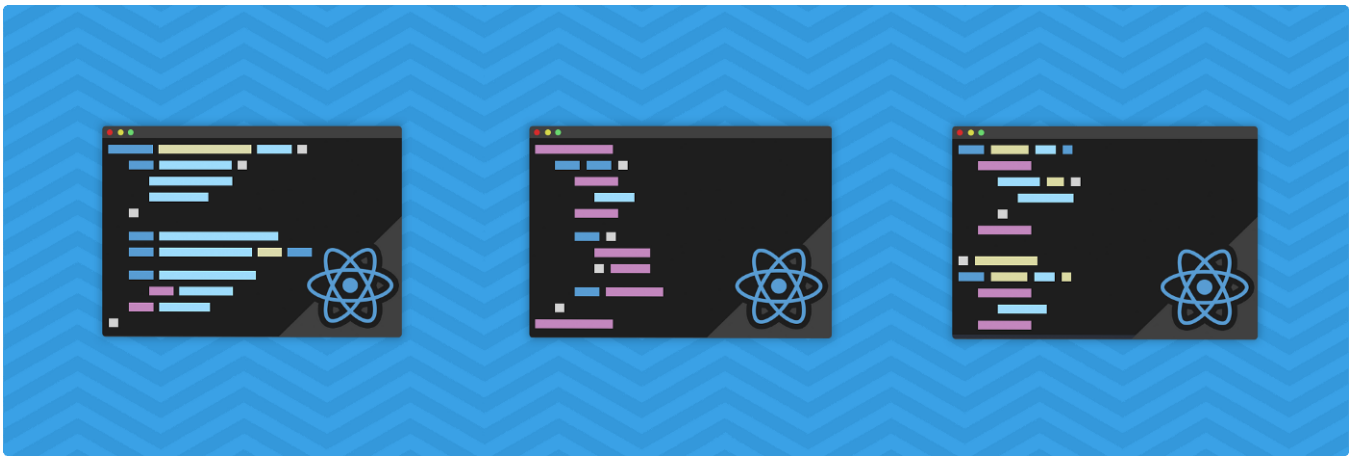
Gustavo Matheus Follow

Oct 26, 2017 · 4 min read · Listen

React Component Patterns

Stateful x Stateless, Container x Presentational, HOCs, Render Callbacks and more

It's been a while since I've been working with **React** — *a Facebook library to build user interfaces using JavaScript* — and there are a few concepts I wish I knew when I was just starting. This text is an attempt to summarize some patterns I learned during my experience so far — and also may be useful for developers who are just about to enter this awesome component-based world.



Stateful x Stateless Components

Just as Stateful and Stateless web services, React components can also hold and manipulate state during application usage (**Stateful**) — or just be a simple component that takes the input props and returns what to display (**Stateless**).

A simple **Stateless** button component that depends on props only:

Gist: stateless.js

And a **Stateful** counter component example (using `Button` component):

```
class ButtonCounter extends React.Component {
  constructor() {
    super()
    this.state = { clicks: 0 }
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    this.setState({ clicks: this.state.clicks + 1 })
  }

  render() {
    return (
      <Button
        onClick={this.handleClick}
        text={`You've clicked me ${this.state.clicks} times!`}
      />
    )
  }
}
```

Gist: stateful.js

As you can see, the last one's constructor holds a component state, while the first one is a simple component that renders a text via props. This separation of concerns may look simple but makes `Button` component highly reusable.

dependencies from the rest of the app, depending only on its own state or props received. Let's take a users list as a **Presentational** component example:

```
const UserList = props =>
  <ul>
    {props.users.map(u => (
      <li>{u.name} – {u.age} years old</li>
    ))}
  </ul>
```

Gist: presentational.js

This list can be updated using our **Container** component:

```
class UserListContainer extends React.Component {
  constructor() {
    super()
    this.state = { users: [] }
  }

  componentDidMount() {
    fetchUsers(users => this.setState({ users }))
  }

  render() {
    return <UserList users={this.state.users} />
  }
}
```

Gist: container.js

This approach divides data-fetching from rendering and also makes `UserList` reusable. If you want to learn more about this pattern, there's an awesome article from Dan Abramov explaining it precisely.

Higher-Order Components

Let's say you need to build an expandable menu component that shows some children content when user clicks on it. So, instead of controlling the state on its parent component, you can simply create a generic **HOC** to handle it:

```
function makeToggleable(Clickable) {
  return class extends React.Component {
    constructor() {
      super()
      this.toggle = this.toggle.bind(this)
      this.state = { show: false }
    }

    toggle() {
      this.setState(prevState => ({ show: !prevState.show }))
    }

    render() {
      return (
        <div>
          <Clickable
            {...this.props}
            onClick={this.toggle}
          />
          {this.state.show && this.props.children}
        </div>
      )
    }
  }
}
```

Gist: hoc.js

This approach allows us to apply our logic to our `ToggleableMenu` component using the JavaScript *decorator* syntax:

```
      <div onClick={this.props.onClick}>
        <h1>{this.props.title}</h1>
      </div>
    )
  }
}
```

Gist: hoc-usage.js

Now we can pass any children to `ToggleableMenu` component:

```
class Menu extends React.Component {
  render() {
    return (
      <div>
        <ToggleableMenu title="First Menu">
          <p>Some content</p>
        </ToggleableMenu>
        <ToggleableMenu title="Second Menu">
          <p>Another content</p>
        </ToggleableMenu>
        <ToggleableMenu title="Third Menu">
          <p>More content</p>
        </ToggleableMenu>
      </div>
    )
  }
}
```

Gist: hoc-menu.js

If you're familiar with Redux's `connect` or React Router's `withRouter` functions, you're already using **HOCs**!

Render Callbacks

Another great way to make a component logic reusable is by turning your component children into a function — that's why **Render Callbacks** are also called **Function as Child Components**. We can take an example of our expandable menu **HOC** and rewrite it using the **Render Callback** pattern:

```
    this.state = { show: false }
  }

  toggle() {
    this.setState(prevState => ({ show: !prevState.show }))
  }

  render() {
    return this.props.children(this.state.show, this.toggle)
  }
}
```

Gist: render-callback.js

Now we can pass a function as our `Toggleable` component children:

```
<Toggleable>
  {(show, onClick) => (
    <div>
      <div onClick={onClick}>
        <h1>First Menu</h1>
      </div>
      { show ?
        <p>Some content</p>
        : null
      }
    </div>
  )}
</Toggleable>
```

Gist: render-callback-usage.js

The code above is already using a function as children, but, if we want to reuse it just as we did in our **HOC** example (multiple menus), we could simply create a new component that uses `Toggleable` logic:

```
          <div onClick={onClick}>
            <h1>{props.title}</h1>
          </div>
          {show && props.children}
        </div>
      )}
    </Toggleable>
```

Gist: render-callback-usage-reusable.js

Our brand new `ToggleableMenu` component is ready to be used:

```
class Menu extends React.Component {
  render() {
    return (
      <div>
        <ToggleableMenu title="First Menu">
          <p>Some content</p>
        </ToggleableMenu>
        <ToggleableMenu title="Second Menu">
          <p>Another content</p>
        </ToggleableMenu>
        <ToggleableMenu title="Third Menu">
          <p>More content</p>
        </ToggleableMenu>
      </div>
    )
  }
}
```

Gist: render-callback-menu.js

Our `Menu` component looks exactly the same as our **HOC** example!

This approach is really useful when we want to change the rendered content itself regardless of **state** manipulation: as you can see, we've moved our **render** logic to our `ToggleableMenu` children function, but kept the **state** logic to our `Toggleable` component!

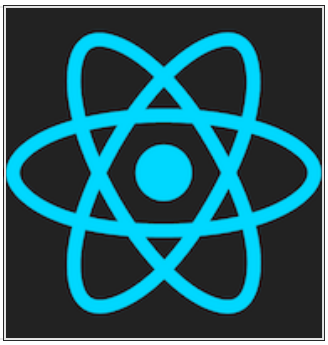
Further Reading

- React Component Patterns by Michael Chan
- React Patterns
- Presentational and Container Components
- React Higher Order Components in depth
- Function as Child Components
- Recompose
- Downshift

Learn React - Best React Tutorials (2019) | gitconnected

The top 45 React tutorials. Courses are submitted and voted on by developers, enabling you to find the best React...

gitconnected.com



Get this newsletter