Advanced React Patterns

in the real world

Lessons from open source community libraries

React Table

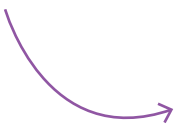
Reach Router

downshift

FORMIK

OHANS EMMANUEL

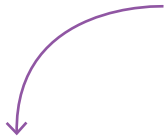
View Author's profile



*List of **patterns** I'll be discussing*

Table of Contents

NB: *Important!*



This 20-page booklet is not an all-inclusive guide but, a reference to the usage of Advanced React Patterns in open-source React community libraries.

See The Complete Guide to Advanced React Patterns for a more complete guide.

1 CUSTOM HOOKS: The Foundational Pattern

Custom Hooks are a mechanism to reuse stateful logic.

NB: name must start with "use"!

EXAMPLE

```
const useAdvancedPatterns = () => {
      // state and effects isolated here
}

// Must be called from a React fn component/other custom hook
useAdvancedPatterns()
```

OPEN-SOURCE EXAMPLES

react-use

TIP: click logo to visit the GIthub page!

Sponsor | Used by 2.6k | Watch 106 | Star 11.9k | Fork 743

React-use exports a set of custom hooks, lots of them, that encapsulate stateful logic relating to UI state, lifecycles, and performing side effects etc.

The use of custom hooks here is as you'd expect; small set of functions that do one thing and help you use the logic embedded in them.

*Examples from **react-use***

- **Lifecycles**
  - `useEffectOnce` — a modified `useEffect` hook that only runs once.
  - `useEvent` — subscribe to events.
  - `useLifecycles` — calls `mount` and `unmount` callbacks.
  - `useMountedState` and `useUnmountPromise` — track if component is mounted.
  - `usePromise` — resolves promise only while component is mounted.
  - `useLogger` — logs in console as component goes through life-cycles.
  - `useMount` — calls `mount` callbacks.
  - `useUnmount` — calls `unmount` callbacks.
  - `useUpdateEffect` — run an `effect` only on updates.

React Table

Sponsor | Used by 17.9k | Watch 126 | Star 10.7k | Fork 1.4k

React-table takes a very interesting approach to custom hooks.

Earlier versions of the library exported a ReactTable component that received 30+ props. While helpful and very customizable, this resulted in a complex and hard to maintain codebase.

The recent version of the library exports no UI component. It exports well designed custom hooks that are meant to be composed by the user to create whatever table UI they've got in mind.

Exported custom hooks from React Table

```
export * from './publicUtils'
export { useTable } from './hooks/useTable'
export { useExpanded } from './plugin-hooks/useExpanded'
export { useFilters } from './plugin-hooks/useFilters'
export { useGlobalFilter } from './plugin-hooks/useGlobalFilter'
export { useGroupBy, defaultGroupByFn } from './plugin-hooks/useGroupBy'
export { useSortBy, defaultOrderByFn } from './plugin-hooks/useSortBy'
export { usePagination } from './plugin-hooks/usePagination'
export { _UNSTABLE_usePivotColumns } from './plugin-hooks/_UNSTABLE_usePivotColumns'
export { useRowSelect } from './plugin-hooks/useRowSelect'
export { useRowState } from './plugin-hooks/useRowState'
export { useColumnOrder } from './plugin-hooks/useColumnOrder'
export { useResizeColumns } from './plugin-hooks/useResizeColumns'
export { useAbsoluteLayout } from './plugin-hooks/useAbsoluteLayout'
export { useBlockLayout } from './plugin-hooks/useBlockLayout'
export { useFlexLayout } from './plugin-hooks/useFlexLayout'
```

PROS

(i) Single Responsibility Modules

As seen in react-use, custom hooks are a simple way to share single responsibility modules within React apps.

(ii) Reduced complexity

Custom hooks are a good way to reduce complexity in your component library. Focus on logic and let the user bring their own UI e.g. React Table.

CONS

(i) Bring your own UI

Historically, most users expect open-source solutions like React Table to include Table UI elements and props to customize its feel and functionality. Providing only custom hooks may throw off a few users. They may find it harder to compose hooks while providing their own UI.

2 THE COMPOUND COMPONENTS PATTERN

The pattern refers to an interesting way to communicate the relationship between UI components and share implicit state by leveraging an explicit parent-child relationship

EXAMPLE

Parent Component

```
<MediumClap>
   <MediumClap.Icon />
   <MediumClap.Total />
   <MediumClap.Count />
</MediumClap>
```

Child Components

```
// Adding the child components to the instance of the Parent
component is completely optional. The following is equally valid

<MediumClap>
   <Icon />
   <Total />
   <Count />
</MediumClap>
```

Child Components

Typically with the Context API

```
// NB: The parent component handles the UI state values and
updates. State values are communicated from parent to child.
```

OPEN-SOURCE EXAMPLES

ReachUI

Sponsor | Watch 52 | Star 3.2k | Fork 335

Consider how an Accordion component from ReachUI is used:

Parent Component

Child Components

```
import {
  Accordion,
  AccordionItem,
  AccordionButton,
  AccordionPanel,
} from “@reach/accordion”;
```

Parent Component

```
function Example() {
  return (
    <Accordion>
      <AccordionItem>
        <h3><AccordionButton> Header </AccordionButton></h3>
        <AccordionPanel>
          Message
        </AccordionPanel>
      </AccordionItem>
    </Accordion>
  );
}
```

which is probably the first thought that comes to mind

As opposed to just exporting a single component, ReachUI exports a parent and accompanying child compoents. This has lots of advantages

```
<Accordion>
   <Accordion.Button />
   <Accordion.Panel />
   <Accordion.Item />
   <Accordion.Item />
</Accordion>
```

these child components can be moved around

or with mutiple identical child components

PROS

(i) Flexible Markup Structure

Users can rearrange the child components in whatever way they seem fit. e.g. having an accordion header at the bottom as opposed to the top.

(ii) Reduced Complexity

As opposed to jamming all props in one giant parent component and drilling those down to child UI components, child props go to their respective child components.

(iii) Separation of Concerns

Having all UI state logic in the Parent component and communicating that internally to all child components makes for a clear division of responsibility.

```
<YourAccordion
    buttonText =`hello`
    buttonPosition =`middle`
    item = {
       [`item #1`, `item#2`]}
    ...evenMoreProps
/>
```

```
<Accordion>
   <Accordion.Button text=`hello`/>
   <Accordion.Panel />
   <Accordion.Item msg=`item#1`/>
   <Accordion.Item msg=`item#2/>
</Accordion>
```

Semantic. Easy to read and understand what goes on internally.

3 EXTENSIBLE STYLES

Regardless of the component you build, a common requirement is allowing the override and addition of new styles.

Allow users style your components like any other element/component in their app.

This is a strong yet rewarding philosophy to work by.

EXAMPLE

```
<YourComponent
  className=`shouldWork`/>

<YourComponent
  style=`shouldWork`/>
```

*As with JSX elements styling via a **className** and **style** prop should be possible*

What about styling via CSS-in-JS solutions?

styled components

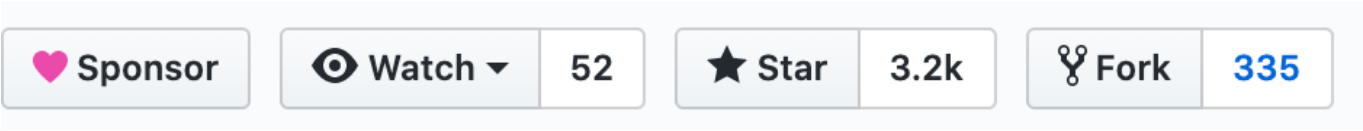
From the docs

Styling any component

The `styled` method works perfectly on all of your own or any third-party component, as long as they attach the passed `className` prop to a DOM element.

*Most CSS-in-JS solutions will work if you handle recieving **className** & **style** props.*

Reach UI



Below's an example of how Reach UI components may be styled. It does a good job of letting users style components as they would other elements in their app.

```
// Emotion and styled components
let YourMenuList = styled(MenuList)`
  border: solid 2px black;
  background: black;
  color: red;
  > [data-reach-menu-item][data-selected] {
    background: red;
    color: white;
  }
`

// normal className
<MenuList className="yep"/>

// normal style
<MenuList style={sure}/>

// glamor CSS prop
<MenuList css={absolutely}/>
```

className prop

style prop

PROS

(i) Intuitive Style Overrides

Allow for style overrides in a way your users are already familiar with.

4 THE CONTROL PROPS PATTERN

Perhaps inspired by React's controlled form elements, control props allow users of your component to control the UI state via certain "control" props.

EXAMPLE

state value passed via props

```
<YourComponent
  value=`string`
  onChange= {fn}/>
```

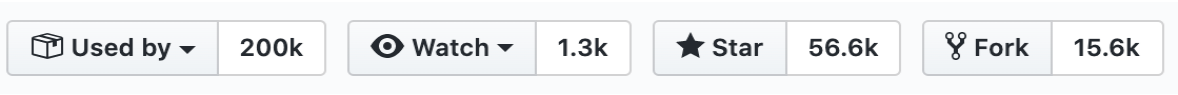
state updater

You'd notice that this is similar to how controlled input elements work in React.

```
<input
  value=`someStateValue`
  onChange= {fnThatUpdatesTheStateValue}/>
```

OPEN-SOURCE EXAMPLES

Material-UI



Consider the Rating component from Material-UI.

Rating component

As seen below, the usage of the component is quite simple.

```
<Rating name=`size-medium` defaultValue={2} />
```

However, if the user of the component wished to control the internal component state, the component allows for controlled props as seen below:

```
<Rating
  value={someValue}
  onChange={someCallback}
  name=`size-medium`
  defaultValue={2} />
```

The rating value will now explicitly be whatever the user sets it to, and every click will invoke the user's state updater, onChange.

```
// click images below to view internal implementation
const { current: isControlled } = React.useRef(valueProp !== undefined);
const [valueState, setValueState] = React.useState(defaultValue);
const valueDerived = isControlled ? valueProp : valueState;
```

*Is state derived from control prop or not? Where **valueProp** is the control prop supplied by the user.*

```
const handleChange = (event) => {
  const newValue = parseFloat(event.target.value);

  if (!isControlled) {
    setValueState(newValue);
  }

  if (onChange) {
    onChange(event, newValue);
  }
};
```

if NOT controlled, call the internal state updater

else call user's callback

PROS

(i) Inversion of Control

A very easy solution to cede control over to the users of your component.

CONS

(i) Duplicate code

For more complex scenarios, the user may have to duplicate some logic you'd have handled internally.

5 THE PROPS COLLECTION PATTERN

Props Collection refer to a collection of common props users of your components/hooks are likely to need.

EXAMPLE

a collection. Typically an object e.g.
{
prop1,
prop2,
prop3
}

```
const {propsCollection} = useYourHook()
```

This is particularly important if you're building a custom hook to be used in conjuction with certain UI elements that are likely to behave in a consistent way.

OPEN-SOURCE EXAMPLES

most open-source solutions prefer to use the more powerful, ***prop getters.*** *Turn to the next page.*

PROS

(i) Ease of Use

This pattern exists mostly for the convenience it brings the users of your component/hooks.

CONS

(i) Inflexible

The collection of props can't be modified or extended.

See the next pattern for a fix.

6 THE PROPS GETTERS PATTERN

Props getters, very much like props collection, provide a collection of props to users of your hooks/component. The difference being the provision of a getter - a function invoked to return the collection of props.

EXAMPLE

A function. When invoked, returns an object e.g.
{
prop1,
prop2,
prop3
}

```
const {getPropsCollection} = useYourHook()
```

The added advantage a prop getter has is it can be invoked with arguments to override or extend the collection of props returned.

```
const {getPropsCollection} = useYourHook()

const propsCollection = getPropsCollection({
  onClick: myClickHandler
  data-testId: `my-test-id`
})
```

user specific values may be passed in.

OPEN-SOURCE EXAMPLES

As seen earlier, React Table provides hooks for building performant tables. To make composing UIs easier, it offers a set of prop getters.

Here's a basic usage of the `useTable` hook from React Table:

```
const {
    getTableProps,
    getTableBodyProps,
  } = useTable({columns,data,})
```

prop getters

a user would then go on to render a table element as follows:

```
<table {...getTableProps()}>
 //other UI elements go here
</table>
```

invoke getter to have props passed to the element

This is an interesting pattern as it allows React Table to provide just custom hooks.

Combine these with prop getters, and users can truly compose whatever UI they seem fit.

```
// Render the UI for your table
return (
  <table {...getTableProps()}>
    <thead>
      {headerGroups.map(headerGroup => (
        <tr {...headerGroup.getHeaderGroupProps()}>
          {headerGroup.headers.map(column => (
            <th {...column.getHeaderProps()}>{column.render('Header')}</th>
          ))}
        </tr>
      ))}
    </thead>
    <tbody {...getTableBodyProps()}>
      {rows.map((row, i) => {
        prepareRow(row)
        return (
          <tr {...row.getRowProps()}>
            {row.cells.map(cell => {
              return <td {...cell.getCellProps()}>{cell.render('Cell')}</td>
            })}
          </tr>
        )
      })}
    </tbody>
  </table>
)
```

count the prop getters in the example

click to view on codesandbox

7 THE STATE INITIALISERS PATTERN

A simple pattern that allows for configurable initial state, and an optional state reset handler.

EXAMPLE

```
const {value, reset} = useYourHook(initialState)
```

user may call this fn to reset state

user passes in some initial state value

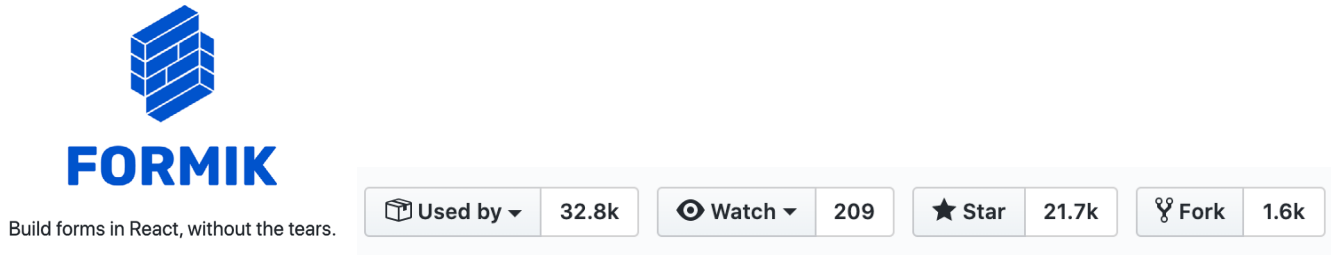
internally

```
const [internalState] = useState(initialState)
```

initialState is passed into your internal state mechanism

Passing props to state is generally frowned upon, which is why you have to make sure the value passed here is only an initialiser. Read on to see examples.

OPEN-SOURCE EXAMPLES



Consider the contrived Formik usage below:

```
import { Formik } from 'formik'

<Formik
      initialValues={{
        firstName: "",
        lastName: "",
        email: ""
      }}
    >

</Formik>
```

state initialiser

```
const initialValues = React.useRef(props.initialValues);
const initialErrors = React.useRef(props.initialErrors || emptyErrors);
const initialTouched = React.useRef(props.initialTouched || emptyTouched);
const initialStatus = React.useRef(props.initialStatus);
const isMounted = React.useRef<boolean>(false);
```

*internally, Formik saves these in a **ref object**, like instance variables. This way subsequent changes are ignored.*

```
const resetForm = React.useCallback(
  (nextState?: Partial<FormikState<Values>>) => {
    const values =
      nextState && nextState.values
        ? nextState.values
        : initialValues.current:
```

*Formik also exposes a **resetForm** callback for users to reset the form state.*

PROS

(i) Important Feature for Most UIs

Setting and resetting state is typically a very important requirement for most UI components. This gives a lot of flexibility to your users.

CONS

(i) May be Trivial

You may find yourself building a component/custom hook where state initialisers are perhaps trivial.

8 STATE REDUCERS

Like the control props pattern, state reducers allow you to cede state control to the users of your component. Also, by leveraging action types, you minimise code duplicates on the user's side.

What??!!

State reducers are perhaps better explained with real examples.

OPEN-SOURCE EXAMPLES

downshift

Used by 9.4k | Watch 89 | Star 8.2k | Fork 661

allows for a stateReducer prop.

```
<Downshift stateReducer={stateReducer}>
```

this is basically a reducer.
const reducer = (state, action) => newState

Each time Downshift sets internal state, the reducer is invoked with the current state and an object that holds the proposed changes. The changes object also includes a "type" that defines the type of action triggering the change.

This allows for interesting use cases e.g. the user may decide to update (or not update) the internal state whichever way they deem fit.

a user's reducer

```
function stateReducer(state, changes) {
  switch (changes.type) {
    case Downshift.stateChangeTypes.keyDownEnter:
    case Downshift.stateChangeTypes.clickItem:
      return {
        ...changes,
        isOpen: state.isOpen,
        highlightedIndex: state.highlightedIndex,
      }
    default:
      return changes
  }
}
```

the library also exports specific types the user can act on!

Maximum user control

PROS

(i) Ultimate Inversion of Control

State reducers in more complicated use cases are the best way to cede control over to the users of your component/custom hooks.

CONS

(i) Complexity

The pattern is arguably the most complex of the bunch to implement.

CONCLUSION ...

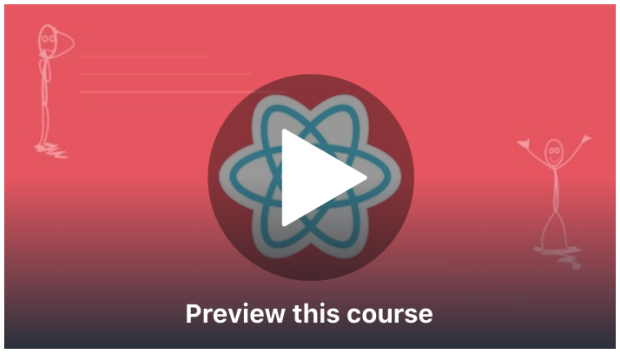
I hope this booklet has shed some light on how popular advanced React component patterns are used by community libraries.

WHAT NEXT ...

If you're new to advanced React patterns, now's the best time to invest in getting up to speed. There's a lot of good materials out there. A simple Google search would lead you to lots of free and paid content!

*I published a pretty good **Udemy course** on the subject.*

Perhaps that interests you ;)



[The Complete Guide to Advanced React Patterns]

All the best!
- Ohans E.