React – Design Patterns

* Presentational Container Component
* HOC – higher order component
* Render Props
* Custom Hooks

Presentational Container Component

The main thing to keep in mind is that we want to

Write our code reusable and open to implementation.

We can only do so if we follow separation of concerns.

React provides us Components and Hooks factory which allows us to create many tiny pieces.

Every piece ( Component / Hook ) should be abstract and reusable , and by combining pieces together we can build new Components on top of each other.

The first two ideas which we would like to separate is logic and View.

We can implement this by creating a Container and a Presentational Component.

Example:

function CounterView({ counter, setCounter }) {

  return (

    <>

      <h3>{counter}</h3>

      <button

        onClick={() => setCounter(prev => prev + 1)}

      >

        Increment

      </button>

    </>

  )

}

The Component above only deals with the view.

It gets the counter and the setCounter from its parent,

The CounterContainer.

function CounterContainer() {

  const [counter, setCounter,] = React.useState(0)

  return (

      <CounterView

        counter={counter}

        setCounter={setCounter}

      />

  )

}

CounterContainer on the other hand does not care about the view, and pass the behavior to the CounterView.

The only problem here is that the CounterView is inside the CounterConatiner, and if we would like to replace the view, well… we need to write the CounterContainer again with another view inside.

Not so reusable…

HOC – Higher Order Component

This pattern allows us to create a Container which can get any Component that we want and wrap it with our logic – such as counter and setCounter.

Exapmle:

function counterHoc(Component) {

  function Counter() {

    const [counter, setCounter,] = React.useState(0)

    return (

      <Component

        counter={counter}

        setCounter={setCounter}

      />

    )

  }

  return Counter

}

In this example, the first thing to notice is that we are using now a regular javascript function instead of a React Component.

This function takes a Component as an argument, which will be the view Component, and the container will pass the behavior to this view Component by its props.

It's important to use capital letter inside the function parameters ('Component'), because we want to use it later on as a React Component.

Inside the function we are declaring another

React Component, and this Component will be the Container which will pass the behavior to the view Component no matter the view.

And finally we are providing the Container in the return statement.

Counter view remain the same:

function CounterView({ counter, setCounter }) {

  return (

    <>

      <h3>{counter}</h3>

      <button

        onClick={() => setCounter(prev => prev + 1)}

      >

        Increment

      </button>

    </>

  )

}

And now we can supply the CounterView

To the counterHoc function

const CounterWithView = counterHoc(CounterView)

 return (

    <>

      <CounterWithView />

    </>

  )

Or any other Component:

 const CounterWithView = counterHoc(({ counter, setCounter }) => (

    <>

      <p>{counter}</p>

      <a href="#"

        onClick={() => setCounter(prev => prev + 1)}

      >

        Increment

      </a>

    </>

  ))

We got more elegant way to implement this idea by using Render props

Render Props means that we can pass the view Component to the container by passing a function which returns the view.

This function will be call inside the container view area,

And we will call this function with the container arguments.

This means we can inject behavior to the function which will return view

Example:

function Counter({ render, }) {

  const [ counter, setCounter, ] = React.useState(0)

  return render(counter, setCounter)

}

CounterView will remain the same:

function CounterView({ counter, setCounter }) {

  return (

    <>

      <h3>{counter}</h3>

      <button

        onClick={() => setCounter(prev => prev + 1)}

      >

        Increment

      </button>

    </>

  )

}

Using Counter:

 <Counter render={( counter, setCounter) => (

        <CounterView counter={counter} setCounter={setCounter} />

      )}/>

Or any other Component:

<Counter render={(counter, setCounter) => (

        <>

          <h3>{counter}</h3>

          <button

            onClick={() => setCounter(prev => prev + 1)}

          >

            Increment

          </button>

        </>

      )} />

We can implement the same idea, but instead of render props, we can use children as a function.

Example:

function Counter({ children, }) {

  const [counter, setCounter,] = React.useState(0)

  return (

    <>

      { children(counter, setCounter) }

    </>

  )

}

Using children:

<Counter>

        {( counter, setCounter ) => (

          <CounterView

            counter={counter}

            setCounter={setCounter}

          />

        )}

</Counter>

Or any other component:

<Counter>

        {(counter, setCounter) => (

          <>

            <h1>{counter}</h1>

            <button onClick={setCounter(pre => pre + 1)}>

              increment

            </button>

          </>

        )}

</Counter>

Until now, every logic container was a React Component.

And as I mention before, we want to separate view and logic.

So if we don’t need any view, it shouldn’t be a React Component at all, it should be a React Hook.

React custom Hooks allows us to use React hooks inside a function, and return a result without using React Component.

The naming convention starts with use and then the function behavior, for instance: useCounter

Example:

export default function useCounter(init) {

    const [ counter, setCounter, ] = React.useState(init)

    function increment(amount) {

        setCounter(prev => prev + amount)

    }

    function decrement(amount) {

        setCounter(prev => prev - amount)

    }

    return { counter, setCounter, increment, decrement, }

}

Using the hook:

function CounterView() {

  const { counter, increment } = useCounter(0)

  return (

    <>

      <h3>{counter}</h3>

      <button

        onClick={() => increment(1)}

      >

        Increment

      </button>

    </>

  )

}