2 - React Design Patterns

React education, 2024.

MITHRIL

Overview

- Compound Pattern
- HOC Pattern
- Hooks Pattern
- Container/Presentational Pattern
- Render Props Pattern

Compound Pattern

Compound Pattern

- The *compound component pattern* allows you to create components that all work together to perform a task.
- Components that are dependent on each other through shared state, and share logic together.
- You often see this with components like <select>, dropdown components or menu items.
- Context API is perfect for implementing this example.

Compound Pattern

- On the right you can see an implementation example with Context API.

```tsx
const FlyOutContext : Context<FlyOutContextType> = createContext<FlyOutContextType>({} as FlyOutContextType)

export function FlyOut({ children }: PropsWithChildren) : Element { Show usages new *
  const [open, toggle] = useState(false)

  return (
    <FlyOutContext.Provider value={{ open, toggle }}>
      {children}
    </FlyOutContext.Provider>
  )
}

function Toggle() : Element { Show usages new *
  const { open, toggle } = useContext(FlyOutContext)

  return (
    <div role="button" onClick={() : void => toggle(!open)}>
      Toggle
    </div>
  )
}

function List({ children }: PropsWithChildren) : false | Element { Show usages new *
  const { open } = useContext(FlyOutContext)
  return open && <ul>{children}</ul>
}

function Item({ children }: PropsWithChildren) : Element { Show usages new *
  return <li>{children}</li>
}

FlyOut.Toggle = Toggle
FlyOut.List = List
FlyOut.Item = Item
```

MITHRIL

Compound Pattern

- On the right you can see usage example of previously made FlyOut component.
- The compound pattern is great when you're building a component library. You'll often see this pattern when using UI libraries like Semantic UI.

MITHRIL

HOC Pattern

HOC Pattern

- Within our application, we often want to use the same logic in multiple components. This logic can include applying a certain styling to components, requiring authorization, or adding a global state.

- One way of being able to reuse the same logic in multiple components, is by using the **higher order component** pattern. This pattern allows us to reuse component logic throughout our application.

- A **Higher Order Component (HOC)** is a component that receives another component. The HOC contains certain logic that we want to apply to the component that we pass as a parameter.

- After applying that logic, HOC returns the element with the additional logic.

HOC Pattern

- Say that we always wanted to add a certain styling to multiple components in our application. Instead of creating style object locally each time, we can simply create a HOC that adds the style objects to the component that we pass to it.
- On the right side example, we just created a StyledButton and StyledText component, which are the modified versions of the Button and Text component. They now both contain the style that got added in the `withStyles` HOC!

MITHRIL

HOC Pattern

- In some cases, we can replace the HOC pattern with React Hooks.
- *“In most cases, Hooks will be sufficient and can help reduce nesting in your tree.”* - React Docs
- By adding a Hook to the component directly, we no longer have to wrap components.
- Using Higher Order Components makes it possible to provide the same logic to many components, while keeping that logic all in one single place. Hooks allow us to add custom behavior from within the component, which could potentially increase the risk of introducing bugs compared to the HOC pattern if multiple components rely on this behavior.

```
<withAuth>
  <withLayout>
    <withLogging>
      <Component />
    </withLogging>
  </withLayout>
</withAuth>
```

MITHRIL

HOC Pattern

- Best use-cases for a HOC:
  - The *same, uncustomized* behavior needs to be used by many components throughout the application.
  - The component can work standalone, without the added custom logic.
- Best use-cases for Hooks:
  - The behavior has to be customized for each component that uses it.
  - The behavior is not spread throughout the application, only one or a few components use the behavior.
  - The behavior adds many properties to the component.

Hooks Pattern

#TBT Class Components

- Before Hooks were introduced in React, we had to use class components in order to add state and lifecycle methods to components. A typical class component in React can look something like

```
class MyComponent extends React.Component {
  /* Adding state and binding custom methods */
  constructor() {
    super()
    this.state = { ... }

    this.customMethodOne = this.customMethodOne.bind(this)
    this.customMethodTwo = this.customMethodTwo.bind(this)
  }

  /* Lifecycle Methods */
  componentDidMount() { ...}
  componentWillUnmount() { ... }

  /* Custom methods */
  customMethodOne() { ... }
  customMethodTwo() { ... }

  render() { return { ... }}
}
```

MITHRIL

Hooks Pattern

- React 16.8 introduced a new feature called Hooks. Hooks make it possible to use React state and lifecycle methods, without having to use a ES2015 class component.
- Although Hooks are not necessarily a design pattern, Hooks play a very important role in your application design. Many traditional design patterns can be replaced by Hooks.
- React Hooks are functions that you can use to manage a components state and lifecycle methods. React Hooks make it possible to:
  - Add state to functional component
  - Manage component's lifecycle
  - Reuse the same stateful logic among multiple components throughout the app

MITHRIL

Hooks Pattern

- Besides the built-in hooks that React provides (`useState`, `useEffect`, `useReducer`, `useRef`, `useContext`, `useMemo`, `useImperativeHandle`, `useLayoutEffect`, `useDebugValue`, `useCallback`), we can easily create our own custom hooks.

- You may have noticed that all hooks start with `use`. It's important to start your hooks with `use` in order for React to check if it violates the rules of Hooks.

Hooks Pattern

- Let’s say we want to keep track of certain keys the user may press when writing the input. Our custom hook should be able to receive the key we want to target as its argument.
- `function useKeyPress(targetKey) {}`

```
export function useKeyPress(targetKey: string) : boolean {
  const [keyPressed, setKeyPressed] = useState(false)

  function handleDown({ key }: KeyboardEvent) : void {
    if (key === targetKey) {
      setKeyPressed(true)
    }
  }

  function handleUp({ key }: KeyboardEvent) : void {
    if (key === targetKey) {
      setKeyPressed(false)
    }
  }

  useEffect(() : () => void => {
    window.addEventListener('keydown', handleDown)
    window.addEventListener('keyup', handleUp)

    return () : void => {
      window.removeEventListener('keydown', handleDown)
      window.removeEventListener('keyup', handleUp)
    }
  }, [])

  return keyPressed
}
```

MITHRIL

Hooks Pattern

- On the right you can see the usage of the made hook.
- Instead of keeping the key press logic local to the `Input` component, we can now reuse the `useKeyPress` hook throughout multiple components, without having to rewrite the same logic over and over.
- Another great advantage of Hooks, is that the community can build and share hooks. We just wrote the `useKeyPress` hook ourselves, but that actually wasn't necessary at all! The hook was already built by someone else and ready to use in our application if we just installed it!

```
export function Input() : Element { Show usages  new *
  const [input, setInput] = useState('')
  const pressQ : boolean = useKeyPress('q')
  const pressW : boolean = useKeyPress('w')
  const pressL : boolean = useKeyPress('l')

  useEffect(() : void => {
    console.log(`The user pressed Q!`)
  }, [pressQ])

  useEffect(() : void => {
    console.log(`The user pressed W!`)
  }, [pressW])

  useEffect(() : void => {
    console.log(`The user pressed L!`)
  }, [pressL])

  return (
    <input
      onChange={(e : ChangeEvent<HTMLInputElement> ) : void => setInput(e.target.value)}
      value={input}
      placeholder="Type something..."
    />
  )
}
```

MITHRIL

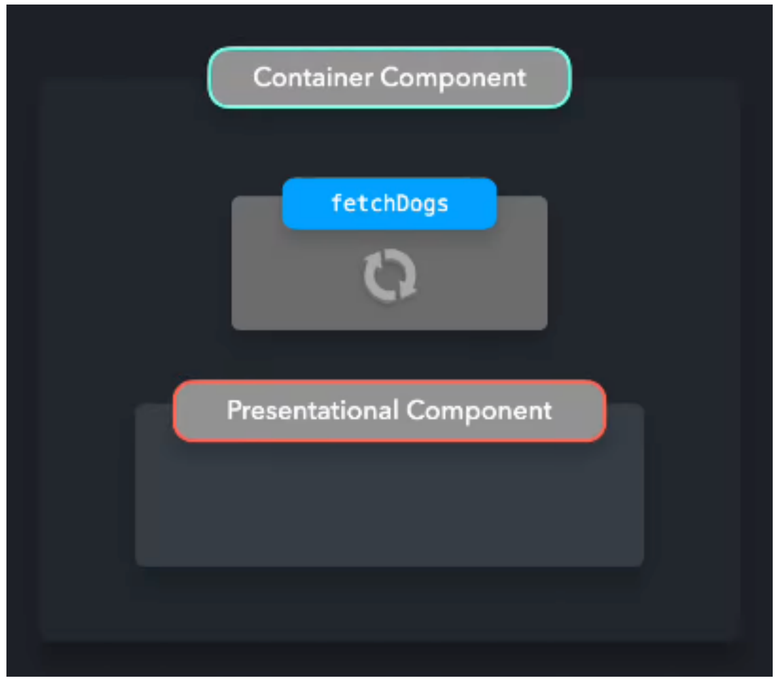
Hooks Pattern

- Some websites that list all the hooks built by the community, and ready to use in your application: useHooks, React Use, Collection of React Hooks
- Pros:
  - Fewer lines of code
  - Simplifies complex components
  - Reusing stateful logic
  - Sharing non-visual logic
- Cons:
  - Have to respect its rules, without a linter plugin, it is difficult to know which rule has been broken.

Container/Presentational Pattern

Container/Presentational Pattern

- In React, one way to enforce separation of concerns is by using the **Container/Presentational pattern**. With this pattern, we can separate the view from the application logic.
- Let's say we want to create an application that fetches 6 dog images, and renders these images on the screen.
- Ideally, we want to enforce separation of concerns by separating this process into two parts:
- **Presentational Components**: Components that care about ***how*** data is shown to the user. In this example, that's the *rendering the list of dog images*.
- **Container Components**: Components that care about ***what*** data is shown to the user. In this example, that's *fetching the dog images*.
- Fetching the dog images deals with **application logic**, whereas displaying the images only deals with the **view**.



MITHRIL

Container/Presentational Pattern

- A presentational component receives its data through props. Its primary function is to simply **display the data it receives** the way we want them to, including styles, *without modifying* that data.
- The `DogImages` component is a presentational component. Presentational components are *usually* stateless: they do not contain their own React state, unless they need a state for UI purposes. The data they receive, is not altered by the presentational components themselves.
- Presentational components receive their data from **container components**.

```
export function DogImages({ dogs }: { dogs: string[] }) : Element[] { Show usages new *
  return dogs.map((dog : string , i : number ) : Element ⇒ <img src={dog} key={i} alt="Dog" />)
}
```

MITHRIL

Container/Presentational Pattern

- The primary function of container components is to **pass data** to presentational components, which they *contain*. Container components themselves usually don't render any other components besides the presentational components that care about their data. Since they don't render anything themselves, they usually do not contain any styling either.
- In our example, we want to pass dog images to the `DogImages` presentational component. Before being able to do so, we need to fetch the images from an external API. We need to create a **container component** that fetches this data, and passes this data to the presentational component `DogImages` in order to display it on the screen.
- Combining these two components together makes it possible to separate handling application logic with the view.

```tsx
export function DogImages({ dogs }: { dogs: string[] }) : Element[] { Show usages new *
  return dogs.map((dog : string , i : number ) : Element => <img src={dog} key={i} alt="Dog" />)
}

export function DogImagesContainer() : Element { Show usages new *
  const [dogs, setDogs] = useState<string[]>([])

  useEffect(() : void => {
    fetch('https://dog.ceo/api/breed/labrador/images/random/6')
      .then((res : Response ) : Promise<any> => res.json())
      .then(({ message } : any ) : void => setDogs(message))
  })

  return <DogImages dogs={dogs} />
}
```

MITHRIL

Container/Presentational Pattern

- In many cases, the Container/ Presentational pattern can be replaced with React Hooks. The introduction of Hooks made it easy for developers to add statefulness without needing a container component to provide that state.
- Instead of having the data fetching logic in the `DogImagesContainer` component, we can create a custom hook that fetches the images, and returns the array of dogs.

```
export function useDogImages() : never[] { Show usages  new *
  const [dogs, setDogs] = useState([])

  useEffect(() : void => {
    fetch('https://dog.ceo/api/breed/labrador/images/random/6')
      .then((res : Response ) : Promise<any> => res.json())
      .then(({ message } : any ) : void => setDogs(message))
  }, [])

  return dogs
}
```

MITHRIL

Container/Presentational Pattern

- By using this hook, we no longer need the wrapping `DogImagesContainer` container component to fetch the data, and send this to the presentational `DogImages` component. Instead, we can use this hook directly in our presentational `DogImages` component!

```
export function DogImages() : Element[] {  Show usages  new *
  const dogs : never[]  = useDogImages()

  return dogs.map((dog : never , i : number ) : Element  => <img src={dog} key={i} alt="Dog" />)
}
```

MITHRIL

Render Props Pattern

Render Props Pattern

- In the section on **Higher Order Components**, we saw that being able to reuse component logic can be very convenient if multiple components need access to the same data, or contain the same logic.

- Another way of making components very reusable, is by using the **render prop** pattern. A render prop is a prop on a component, which value is a function that returns a JSX element. The component itself does not render anything besides the render prop. Instead, the component simply calls the render prop, instead of implementing its own rendering logic.

Render Props Pattern

- Imagine that we have a Title component. In this case, the Title component shouldn't do anything besides rendering the value that we pass. We can use a render prop for this! Let's pass the value that we want the Title component to render to the render prop.
- The cool thing about render props, is that the component that receives the prop is very reusable. We can use it multiple times, passing different values to the render prop each time.
- Although they're called *render* props, a render prop doesn't have to be called `render`. Any prop that renders JSX is considered a render prop!

```
const Title = (props) => props.render();
```

```
<Title render={() => <h1>I am a render prop!</h1>} />
```

MITHRIL

Render Props Pattern

- We've just seen that we can use render props in order to make a component reusable, as we can pass different data to the render prop each time. But, why would you want to use this?
- A component that takes a render prop usually does a lot more than simply invoking the `render` prop. Instead, we usually want to pass data from the component that takes the render prop, to the element that we pass as a render prop!

```
function Component(props) {
  const data = { ... }

  return props.render(data)
}
```

```
<Component render={data => <ChildComponent data={data} />}
```

MITHRIL

Render Props Pattern

- Besides regular JSX components, we can pass functions as children to React components. This function is available to us through the `children` prop, which is technically also a render prop.
- We have access to this function, through the `props.children` prop that's available on the `Input` component. Instead of calling `props.render` with the value of the user input, we'll call `props.children` with the value of the user input.

```
function Input(props) {
  const [value, setValue] = useState("");

  return (
    <>
      <input
        type="text"
        value={value}
        onChange={(e) => setValue(e.target.value)}
        placeholder="Temp in °C"
      />
      {props.children(value)}
    </>
  );
}
```

MITHRIL

Render Props Pattern

- Since we explicitly pass props, we solve the HOC's implicit props issue. The props that should get passed down to the element, are all visible in the render prop's arguments list. This way, we know exactly where certain props come from.

- Since we can't add lifecycle methods to a `render` prop, we can only use it on components that don't need to alter the data they receive.