React.5

patterns

https://reactpatterns.js.org/docs/

▼ What is a proxy component?

A proxy component is a placeholder component that can be rendered to or from another component.

In short a proxy component is a **reusable component**.

```
import React from 'react'

class Button extends React.Component {
  render() {
    return <button type="button">My Button</button>
  }
}

class App extends React.Component {
  render() {
    return <Button />
  }
}

export default App
```

▼ Make the API Call in the useEffect hook

you can use setState to update your component when the data is retrieved

```
useEffect(() => {
  fetch('api/sms')
    .then(result => {
      const sms = result.data
      console.log('COMPONENT WILL Mount messages : ', sms)
      this.setState({sms: [...sms.content]})
    })
}, [])
```

▼ What is a stateless function?

a way to define React **presentational** components as a function

Stateless function does not hold state; just props

```
const UserPassword = function(props) {
  return <p>The user password is: {this.props.userpassword}</p>
};
```

▼ What is a Higher-Order Function?

Functions that operate on other functions, either by taking them as arguments or by returning them are called higher-order functions.

```
function unless(test, then) {
  if (!test) then()
}

repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, "is even")
  })
})
// 0 is even
// 2 is even
```

▼ What is a Higher-Order Component?

A higher-order component is a function that takes a component and returns a new component.

```
const withColor = Element => props => <Element {...props} color="red" />

const Button = () => {
  return <button>My Button</button>
}

const ColoredButton = withColor(Button)
```

▼ How to access a child's DOM node from a parent component?

through Refs

```
import React from 'react'

class Input extends React.Component {
  constructor(props) {
    super(props)
```

```
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef()
  }

  focus() {
    // EXPLANATION: a reference to the node becomes accessible at the current attribute of the ref.
    // make the DOM node focus
    this.textInput.current.focus();
  }

  render() {
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    )
  }
}
```

▼ What are JSX spread attributes?

it's a syntax for passing all of an object's properties as JSX attributes

```
const component = <Component {...props} /
```

▼ What is a render callback?

For example below, notice that we create a function foo which takes a callback function as a parameter.

When we call foo, it turns around and calls back to the passed-in function.

```
const foo = (hello) => {
  return hello('foo')
}

foo((name) => {
  return `hello from ${name}`
})
```

▼ What is Function as a Child Component?

A Function as child component is a pattern that lets you pass a render function to a component as the children prop so **you can change what you can pass as children to a component**.

```
import React from 'react'

class PageWidth extends React.Component {
```

```
  state = { width: 0 }

  componentDidMount() {
    this.setState({ width: window.innerWidth })

    window.addEventListener(
      'resize',
      ({ target }) => {
        this.setState({ width: target.innerWidth })
      }
    )
  }

  render() {
    const { width } = this.state

    return this.props.children(width)
  }
}

<PageWidth>
  {width => <div>Page width is {width}</div>}
</PageWidth>
```

▼ What is Function as a Prop Component?

this is how you can pass a function as a prop to a component

```
const hello = (name) => {
  return <div>`hello from ${name}`</div>
}

const Foo = ({ hello }) => {
  return hello('foo')
}

<Foo hello={hello} />
```

▼ What is Component Injection?

Passing (or inject) a component into another component it's called Component Injection.

```
const Hello = ({ name }) => {
  return <div>`hello from ${name}`</div>
};

const Foo = ({ Hello }) => {
  return <Hello name="foo" />
};
```

▼ Conditional rendering: if-else

```
function List({ list }) {
  if (!list) {
    return null
  }

  if (!list.length) {
    return <p>Sorry, the list is empty.</p>
  } else {
    return (
      <div>
        {list.map(item => <ListItem item={item} />)}
      </div>
    )
  }
}
```

▼ Conditional rendering: Ternary operation

```
function Item({ item, mode }) {
  const isEditMode = mode === 'EDIT'

  return (
    <div>
      { isEditMode
          ? <ItemEdit item={item} />
          : <ItemView item={item} />
      }
    </div>
  )
}
```

▼ Conditional rendering: Logical && operator

```
function LoadingIndicator({ isLoading }) {
  return (
    <div>
      { isLoading && <p>Loading...</p> }
    </div>
  )
}
```

▼ Conditional rendering: Switch case

```
function Notification({ text, state }) {
  switch(state) {
    case 'info':
      return <Info text={text} />
    case 'warning':
      return <Warning text={text} />
    case 'error':
```

```
      return <Error text={text} />
    default:
      return null
  }
}

Notification.propTypes = {
  text: React.PropTypes.string,
  state: React.PropTypes.oneOf(['info', 'warning', 'error'])
}
```

▼ Conditional rendering with enum

```
const NOTIFICATION_STATES = {
  info: <Info />,
  warning: <Warning />,
  error: <Error />,
}

function Notification({ state }) {
  return (
    <div>
      {NOTIFICATION_STATES[state]}
    </div>
  )
}
```

▼ Conditional rendering: With Higher Order Component HOC

e.g. one use case could be to alter the look of a component

```
// HOC declaration
function withLoadingIndicator(Component) {
  return function EnhancedComponent({ isLoading, ...props }) {
    if (!isLoading) {
      return <Component { ...props } />
    }

    return <div><p>Loading...</p></div>
  }
}

// Usage
const ListWithLoadingIndicator = withLoadingIndicator(List)

<ListWithLoadingIndicator
  isLoading={props.isLoading}
  list={props.list}
/>
```

▼ What is destructuring?

Destructuring is a JavaScript expression that allows us to extract data from arrays, objects, and maps and set them into new, distinct variables.

▼ Destructuring the objects this.props and this.state.

```
class ChainedModals extends Component {
  render() {
    const { modalList } = this.props
    const { currIndex, showModal } = this.state

    // ...
  }
}
```

▼ What does the ... rest operator do? (props pattern)

it gathers the rest of the items in the object and puts them in a variable

```
function Modal(props) {
  var onClick = props.onClick
  var show = props.show
  var backdrop = props.backdrop

  return (
    <Modal show={show} backdrop={backdrop}>
      <Button onClick={onClick}>Next</Button>
    </Modal>
  )
}

// INSTEAD:

function Modal({ onClick, ...rest }) {
  return (
    <Modal {...rest}>
      <Button onClick={onClick}>Next</Button>
    </Modal>
  )
}
```

▼ For HTTP Request use what over callbacks?

promises

▼ Container Component (Stateful Component) pattern?

A container does data fetching and then renders its corresponding sub-component.

```
class CommentListContainer extends React.Component {
  state = { comments: [] }

  componentDidMount() {
    fetchSomeComments(comments =>
      this.setState({ comments: comments }))
  }

  render() {
    return <CommentList comments={this.state.comments} />
  }
}

const CommentList = props =>
  <ul>
    {props.comments.map(c => (
      <li>{c.body}—{c.author}</li>
    ))}
  </ul>
```

▼ What is the State Hoisting (pattern)?

Create a common ancestor and in this common ancestor then use the state to manage all the data and callbacks that children will use in rendering

▼ What is Pure Component?

PureComponent changes the life-cycle method shouldComponentUpdate and adds some logic to automatically check whether a re-render is required for the component.

```
import { PureComponent } from 'react'

export default class MyComponent extends PureComponent {
  // Won't re-render when the props DONT change
  render() {
    return <SomeComponent someProp={props.someProp}/>
  }
}
```

▼ Anti-pattern: Props in Initial State

```
// don't do this

constructor(props) {
    super(props)
    this.state = {
      confirmed: false,
      inputVal: props.inputValue
    }
  }
```

▼ Anti-pattern: shouldComponentUpdate Avoid Heavy Re-render

we want to check if the props that we use in this component have changed

```
// do this to fight the anti-pattern

export default class AutocompleteItem extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return nextProps.url !== this.props.url || nextProps.selected !== this.props.selected
  }
...
}
```

▼ Anti-pattern: Indexes as a Key

Keys should be unique so that React can keep a better track of elements.

What going on if you push an item to the list or remove items in the middle, if the key is same as before React assumes that the DOM element represents the same component as before.

```
// do this instead of indices

{todos.map((todo) =>
  <Todo
    {...todo}
    key={todo.id}
  />
)}
```

▼ Anti-pattern: Spreading props on DOM Elements. Bad practice.

When we spread props into a Dom element, we run the risk of adding unknown HTML attributes, which is a bad practice.

```
// don't do this
const Spread = props => <div {...props} />

// do this
<Spread className="foo" />
```