Level up with React

Rishita Narayan
Developer
rishita.narayan@thoughtworks.com

/thoughtworks

Agenda

Prerequisites

- Basic Knowledge of HTML, CSS, JavaScript, VS Code
- CodeSandBox account
- React Developer Tools extension installed

/thoughtworks

React

/thoughtworks

React is a declarative JavaScript library for building user interfaces.

It lets you compose complex UIs from small isolated and reusable pieces of code called “components”.

/thoughtworks

Popular Sites which uses React

Netflix

Facebook

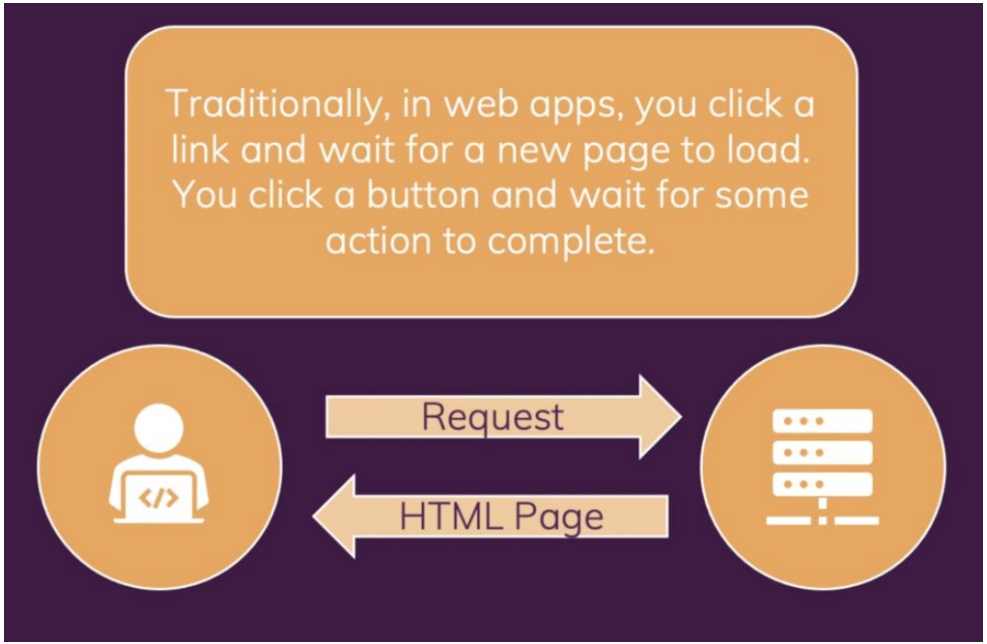
/thoughtworks

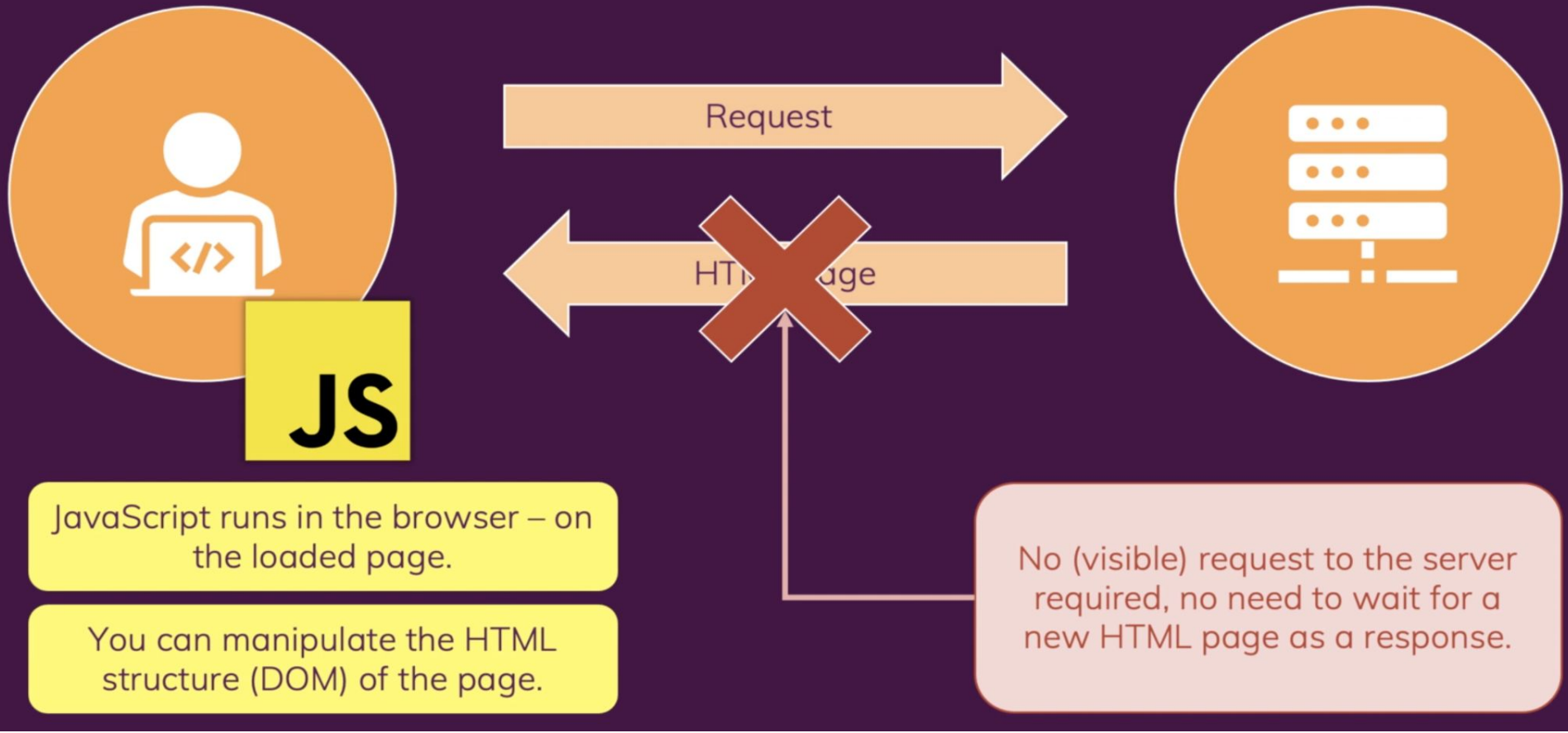
How React Works

/thoughtworks

Traditional Web Apps

- Browser sends request to server where your code is hosted
- The server replies with the html response which the browser should display
- You need a network connection to send and receive the response from server
- Website might have latency where you wait for new html page to be returned from server





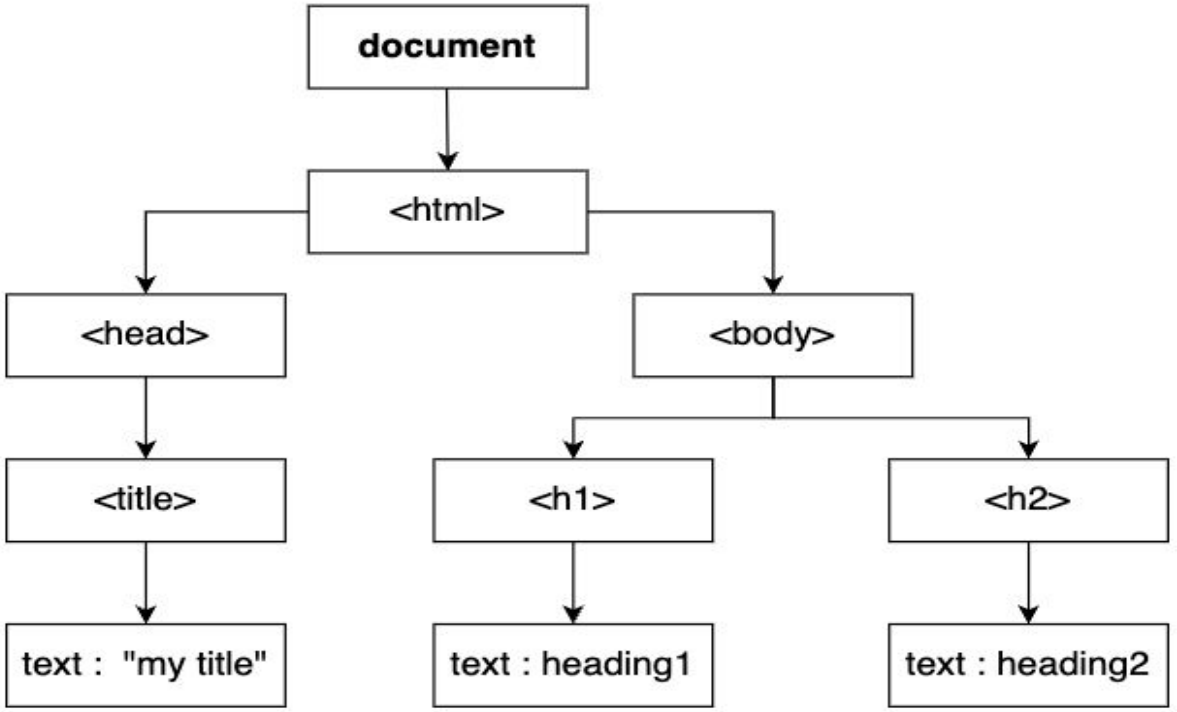
Understanding the DOM

/thoughtworks

DOM stands for "Document Object Model". It represents the entire UI of the web application as a tree data structure.

In simpler terms, it is a structural representation of the HTML elements of the web application.

/thoughtworks



Browser DOM tree

/thoughtworks

Understanding DOM

- DOM Manipulation is the heart of the modern, **interactive web**. Unfortunately, it is also a **lot slower** than most JavaScript operations.
- This **slowness** is made worse by the fact that most JavaScript frameworks **update the DOM** much more than they have to.
- Let's say that you have a list that contains ten items. You check off the first item. Most JavaScript frameworks would rebuild the entire list. That's ten times more work than necessary! Only one item changed, but the remaining nine get rebuilt exactly how they were before.

Why?

The more UI components you have on your page, the more expensive the DOM updates can be, since they would all be re-rendered on every DOM update.

Virtual DOM

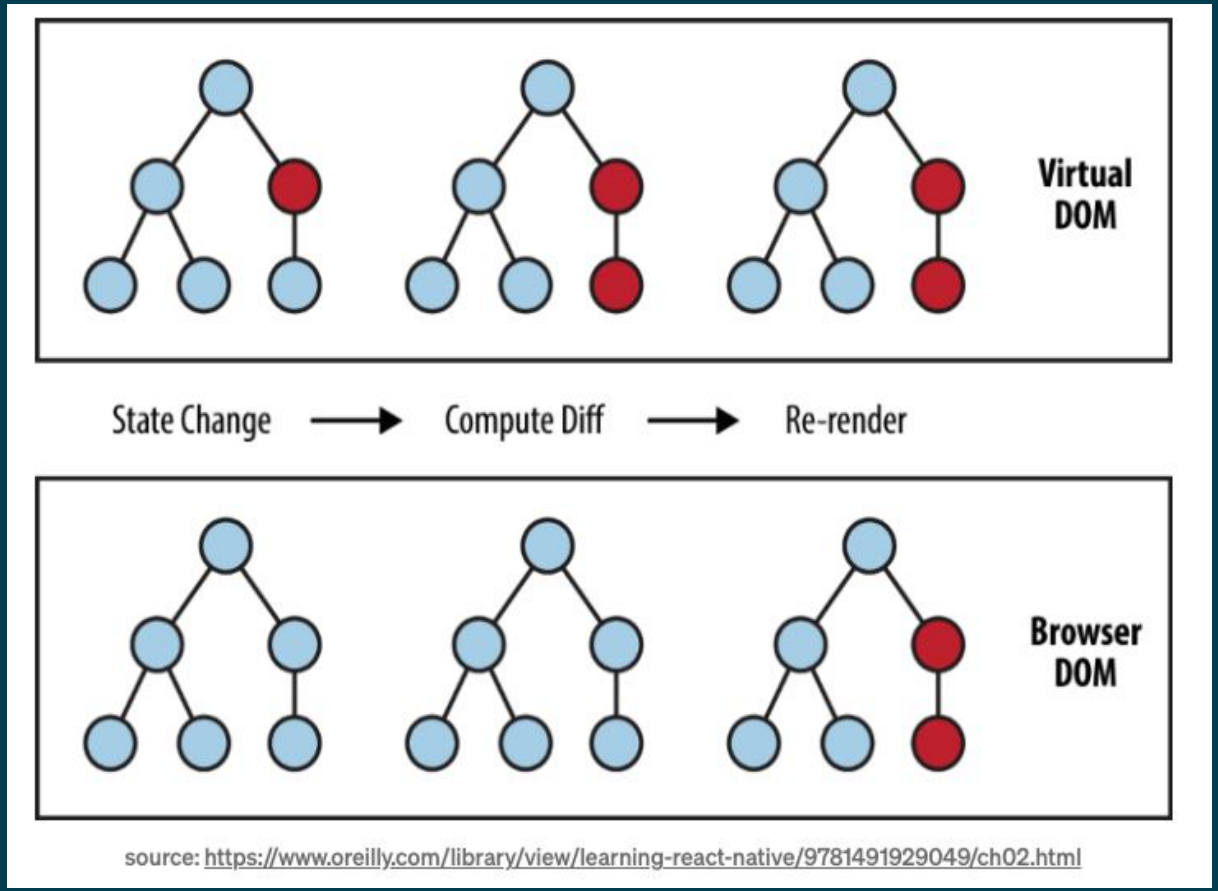
- Like the actual DOM, the **Virtual DOM is just a node tree** that lists elements and their attributes and content as objects and properties.
- When new elements are added to the UI, a fresh virtual DOM is created with each element being a node on the tree.
- When you render a JSX element, every single virtual DOM object gets updated.
- React monitors the values of each component's state with the Virtual DOM. When a component's state changes, React compares the existing DOM state with what the new DOM should look like. After that, it finds the least expensive way to update the DOM.

Why?

The React team realized that JavaScript is fast, but updating the DOM makes it slow. **React minimizes DOM changes(the most expensive change).**

React finds the least expensive way to update the DOM.

Instead of sending updates for each single change in state, these are sent over to the DOM in **one batch to ensure that re-painting event is only performed once**.

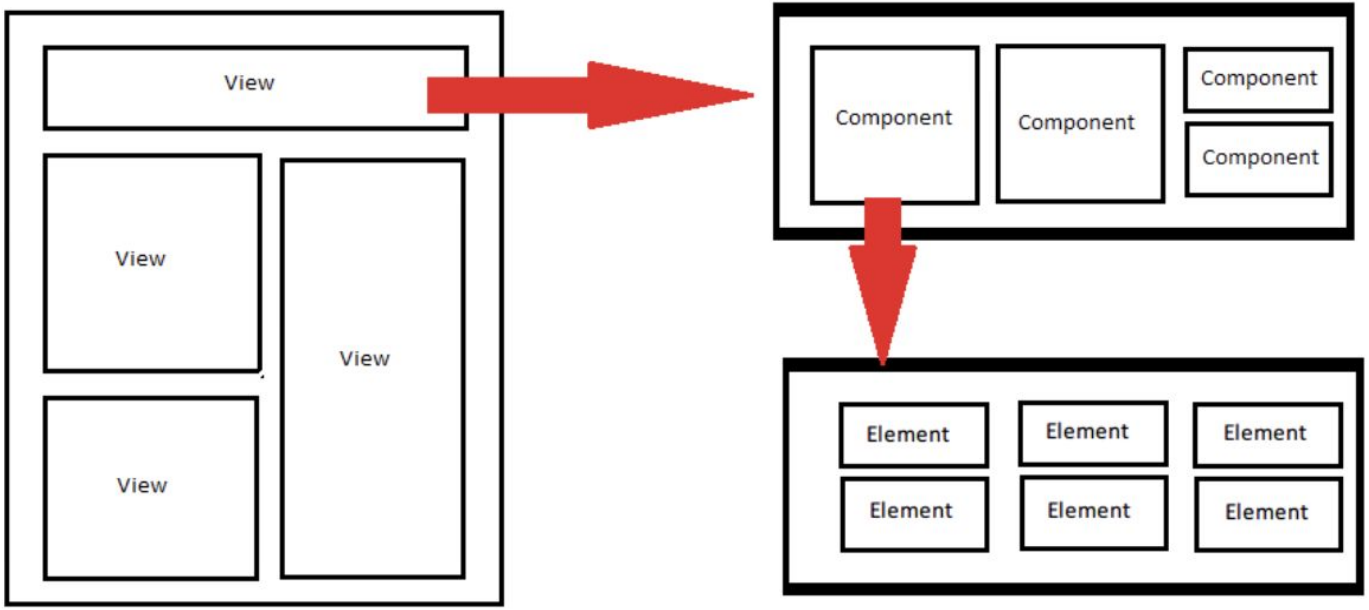


source: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html

/thoughtworks

Components

/thoughtworks



React js Building Blocks

/thoughtworks

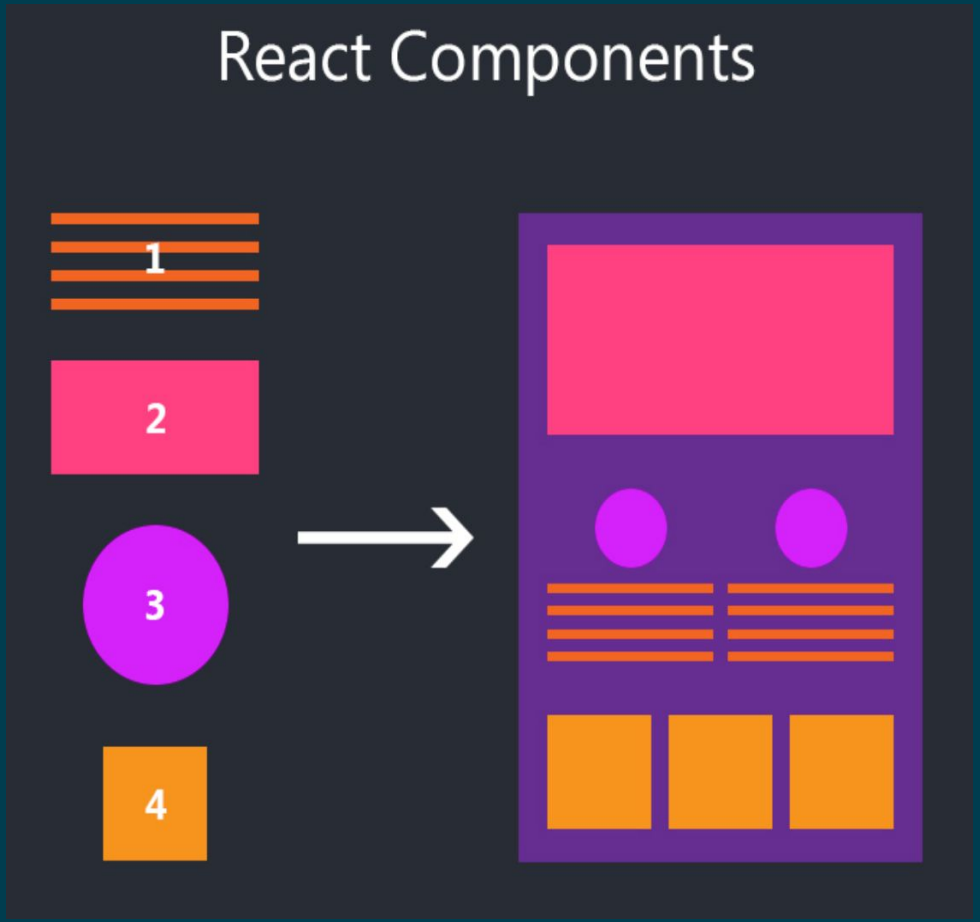
Components let you split the UI into **independent, reusable** pieces, and think about each piece in **isolation**.

Components are like JavaScript functions. They accept arbitrary inputs (called “props”) and **return React elements** describing what should appear on the screen.

With components we achieve

- Reusability
- Code abstraction
- Code isolation

/thoughtworks



JSX

React uses a special syntax known as JavaScript XML (JSX).

JSX allows you to **integrate both HTML and JavaScript** into a single file or even single line of code.

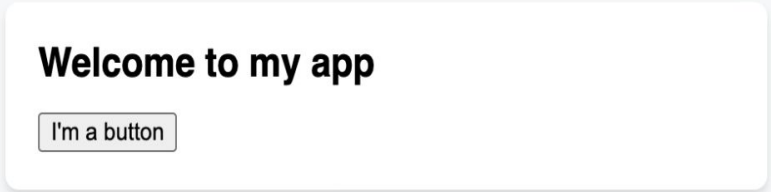
Browsers don't natively support JSX. So JavaScript and HTML must be generated from the JSX files to be rendered by a browser.

Whenever we write React Code, we always build/**transpile** it to JavaScript that browsers can understand. This is generally done with tools like Babel or TypeScript.

```
1  const user = {
2    name: 'Hedy Lamarr',
3    imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
4    imageSize: 90,
5  };
6
7  export default function Profile() {
8    return (
9      <>
10       <h1>{user.name}</h1>
11       <img
12         className="avatar"
13         src={user.imageUrl}
14         alt={'Photo of ' + user.name}
15         style={{
16           width: user.imageSize,
17           height: user.imageSize
18         }}
19       />
20     </>
21   );
22 }
23
```

Component

```
1  function MyButton() {
2    return (
3      <button>
4        I'm a button
5      </button>
6    );
7  }
8
9  export default function MyApp() {
10   return (
11     <div>
12       <h1>Welcome to my app</h1>
13       <MyButton />
14     </div>
15   );
16 }
17
```



Notice that <MyButton /> starts with a capital letter.

That's how you know it's a React component.

Composing Component

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

we can create an App component that renders Welcome many times

.

•

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

Some components don’t know their children ahead of time. We can use the special **children** prop to pass children.

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

Responding to Events

You can respond to events by declaring event handler functions inside your components

Notice how onClick={handleClick} has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

```
function MyButton() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

.

Passing data to components

PROPS

We use props in React to pass data from one component to another (**from a parent component to a child component(s)**).

Props is just a shorter way of saying properties.

They are useful when you want the flow of data in your app to be dynamic.

Components are Pure

Props are READ ONLY

Whether you declare a component as a function or a class, it must never modify its own props. Consider this `sum` function:

```
function sum(a, b) {
  return a + b;
}
```

Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.

All React components must act like pure functions with respect to their props.

In contrast, this function is impure because it changes its own input

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

If Components can’t change Props how do we change data?

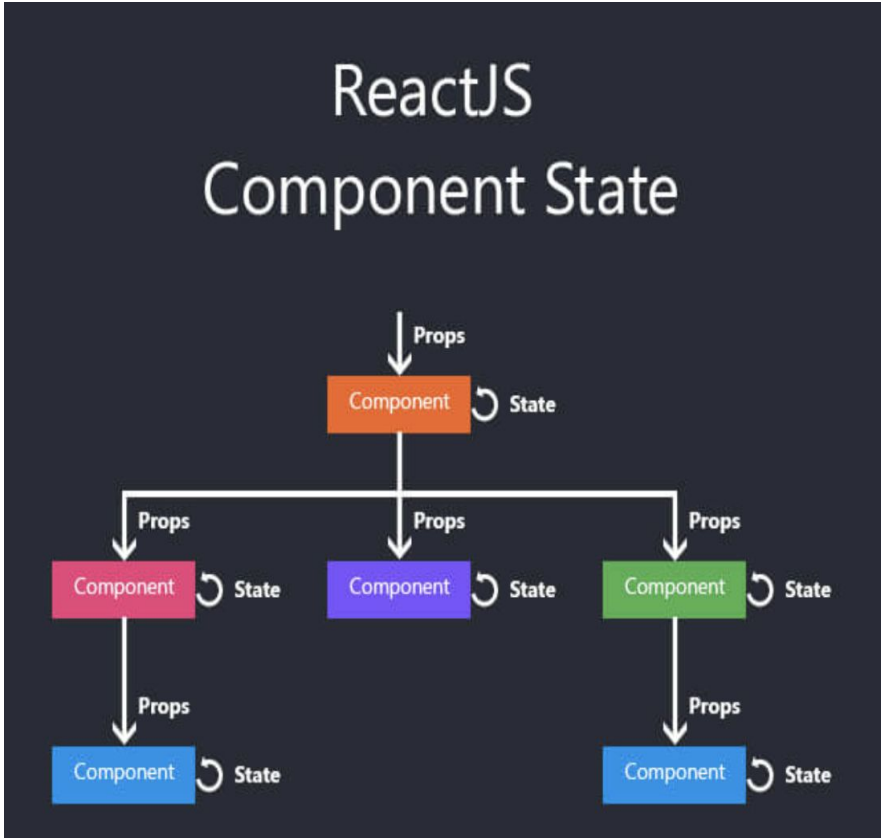
/thoughtworks

State

The state is a built-in React object that is used to contain data about the component.

A component's state can change over time; whenever it changes, the component re-renders.

The change in state can happen as a response to user action or system-generated events.



```jsx
import { useState } from 'react';

function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}

```

With `setCount`, you're telling React:

- Replace `count`with this new object
- And render this component again

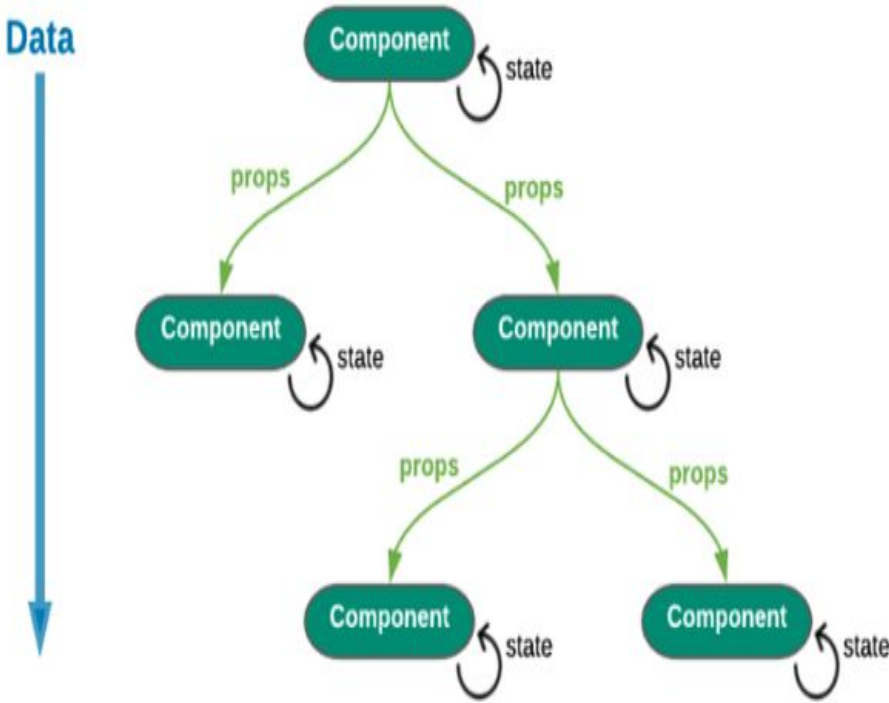
The Data Flows Down

Each component minds its own business

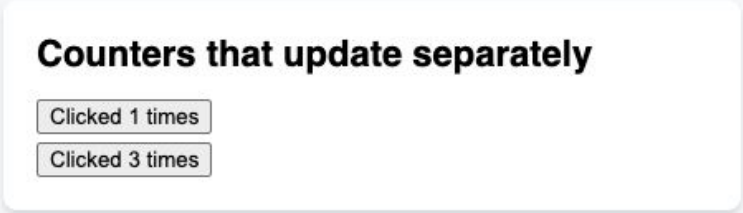
Parent component can choose to send props to its child component but doesn't know anything else about its children

parent/child components can not know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

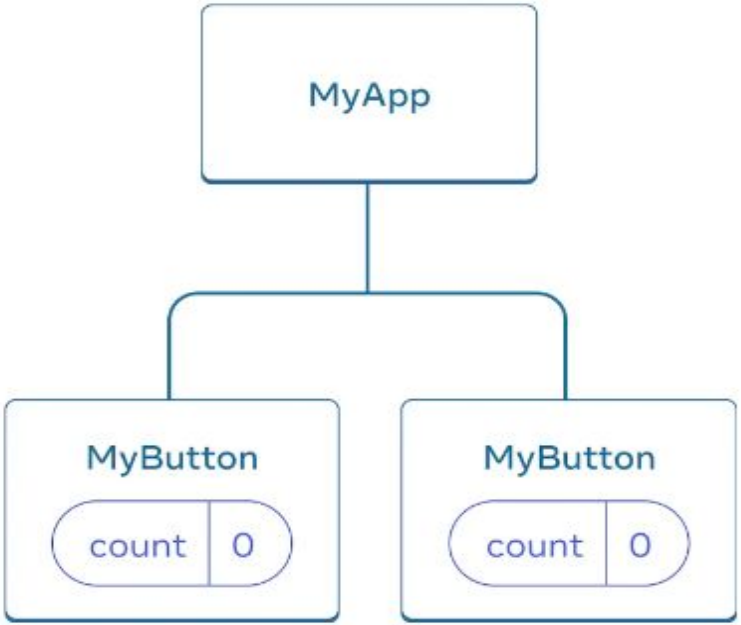
This is why state is often called local or encapsulated.



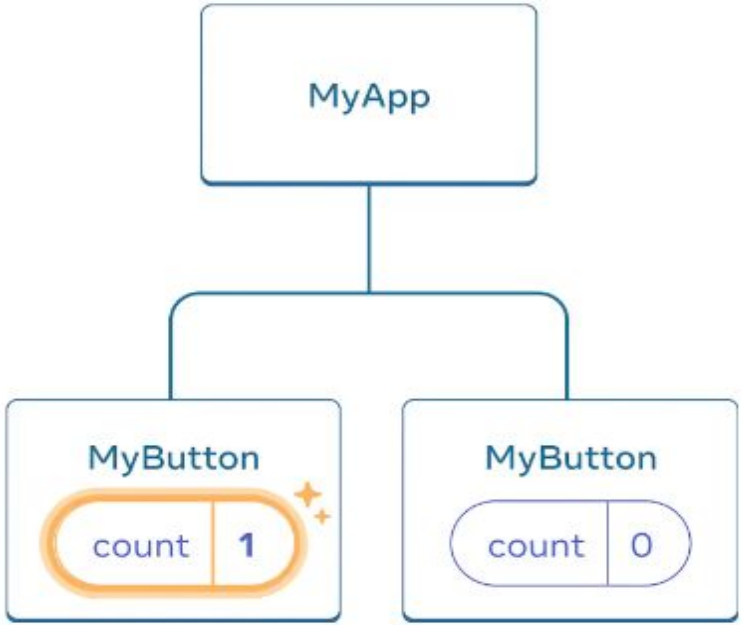
```jsx
import { useState } from 'react';

function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

```



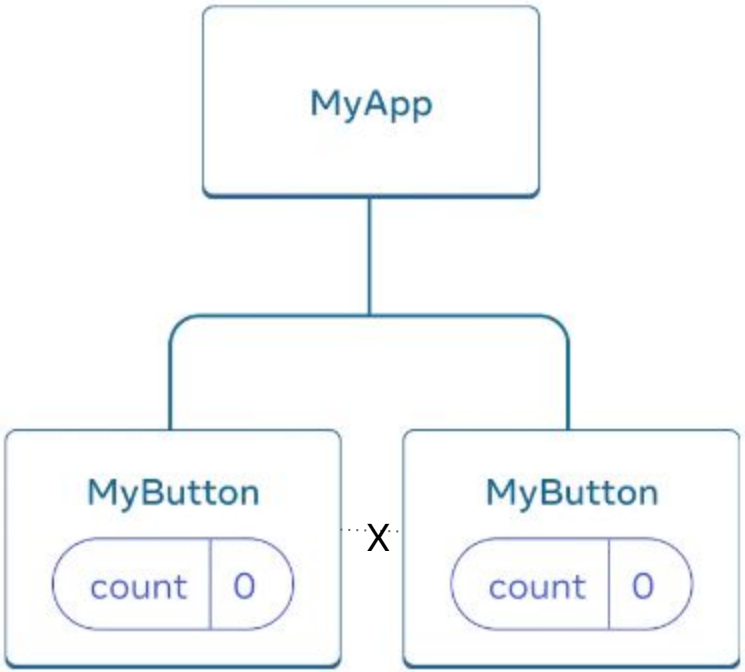
CodeSandBox

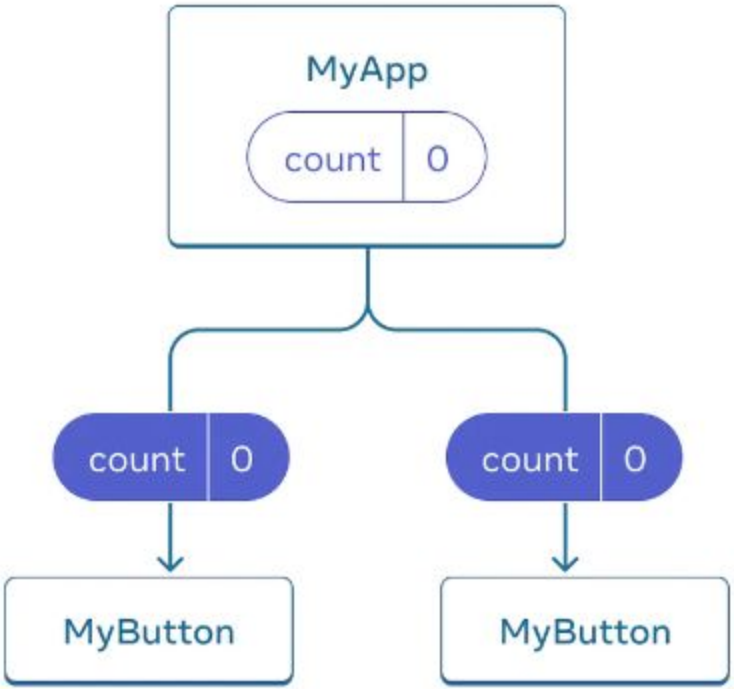


Initially, each `MyButton`'s `count` state is `0`

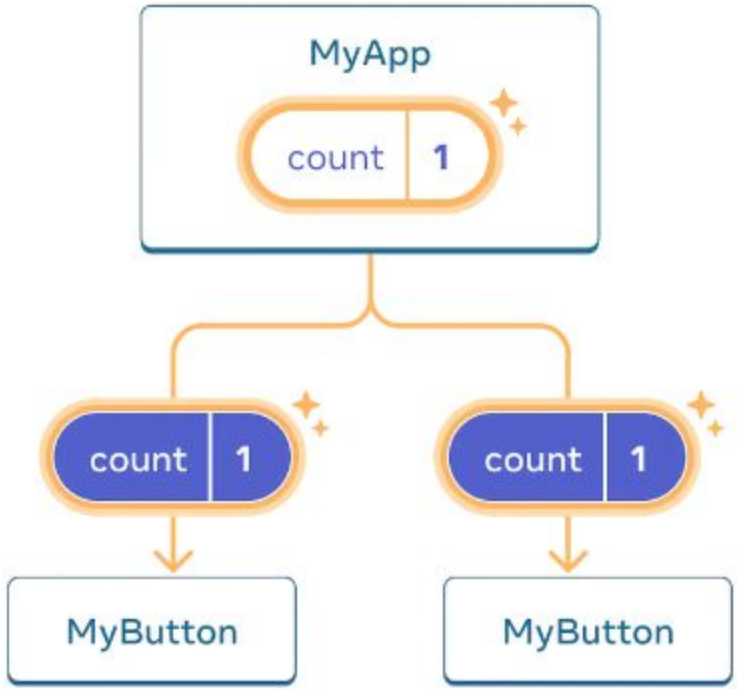


The first `MyButton` updates its `count` to `1`





Initially, `MyApp`'s `count` state is `0` and is passed down to both children



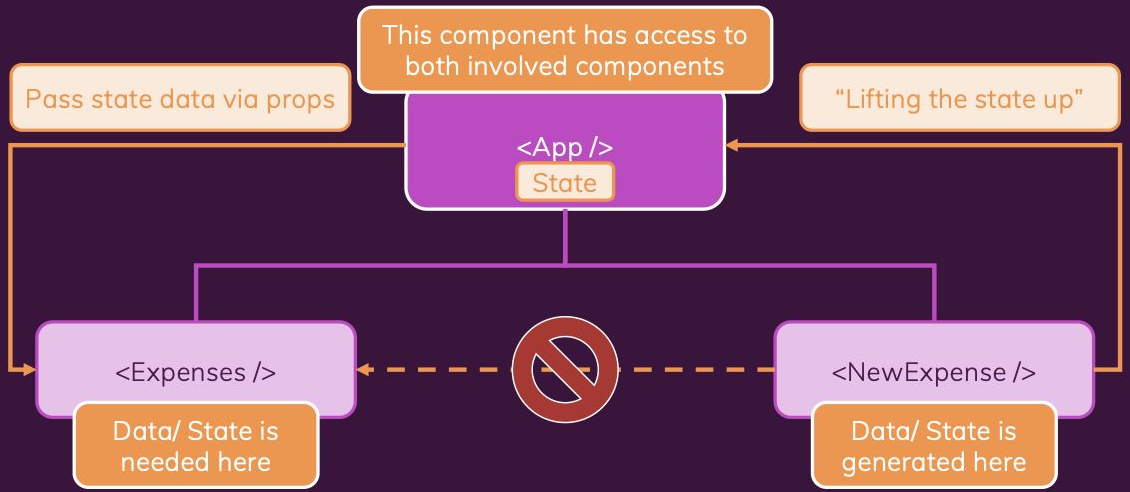
On click, `MyApp` updates its `count` state to `1` and passes it down to both children

```jsx
import { useState } from 'react';

function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}

export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}
```

```jsx
import { useState } from 'react';

function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}
```

ACADEMIND

Lifting State Up



Props

- Props can only pass from **parent component to child component**.
- Parent component can send props to multiple children components.
- Props are **read-only**. They cannot be modified.
- Sibling **can’t send props to sibling componen**t.
- **Child component can’t directly send props to a parent component**; a parent has to use a callback function to receive the data from the child component.

State

- State is mutable.
- We can pass an object or a function in setState
- State can be private or public to its children components.
- It **automatically re-rerenders whenever there is a change in their state**.
- No guarantee about its synchronous performance.

Understanding Lifecycle

/thoughtworks

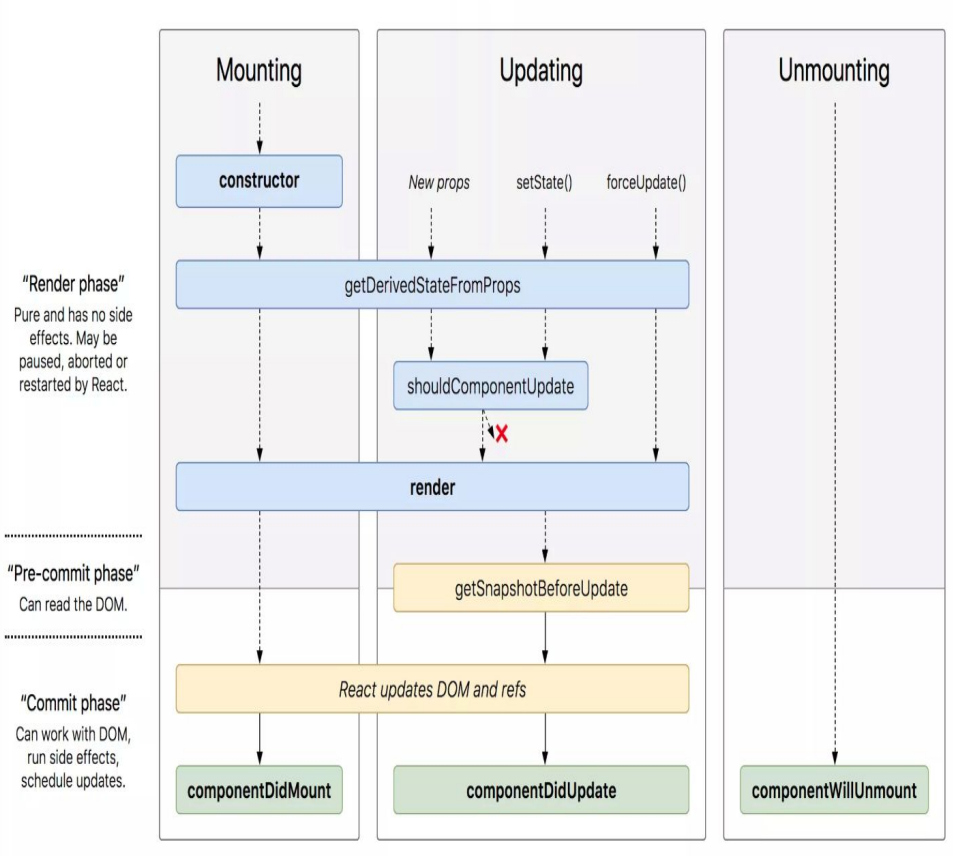
Lifecycle

You can think of React lifecycle methods as the series of events that happen from the birth of a React component to its death.

Every component in React goes through a lifecycle of events. I like to think of them as going through a cycle of birth, growth, and death.

- **Mounting** – Birth of your component
- **Update** – Growth of your component
- **Unmount** – Death of your component

Now that we understand the series of lifecycle events let's learn more about how they work.



source

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

```jsx
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

Understanding Hooks

/thoughtworks

Understanding Hooks

- Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.
- They let you use state and other React features without writing a class.
- With Hooks, you can extract stateful logic from a component so it can be tested independently and reused.
- Hooks allow you to reuse stateful logic without changing your component hierarchy.
- This makes it easy to share Hooks among many components or with the community.

Why?

It’s hard to reuse stateful logic between components

Complex components become hard to understand

Why Hooks

- **It's hard to reuse stateful logic between components**

  With Hooks, you can extract stateful logic from a component so **it can be tested independently and reused**. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components

.

- **Complex components become hard to understand**

  For example, components might perform some data fetching in `componentDidMount` and `componentDidUpdate`. However, the same `componentDidMount` method might also contain some unrelated logic that sets up event listeners, with cleanup performed in `componentWillUnmount`. **Mutually related code that changes together gets split apart, but completely unrelated code ends up combined in a single method**. This makes it too easy to introduce bugs and inconsistencies.

useState

Call `useState` at the top level of your component to declare one or more state variables.

The convention is to name state variables like `[something, setSomething]` using array destructuring.

`useState` returns an array with exactly two items:

1. The current state of this state variable, initially set to the initial state you provided.
2. The `set` function that lets you change it to any other value in response to interaction.

```
import { useState } from 'react';

function MyComponent() {
  const [age, setAge] = useState(42);
  const [name, setName] = useState('Taylor');
  // ...
```

useEffect

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.

When you call `useEffect`, y**ou're telling React to run your "effect"** function after flushing changes to the DOM. Effects are declared inside the component so they have access to its props and state. By default, React runs the effects after every render — *including* the first render.

If you're familiar with React class lifecycle methods, you can think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined.

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Write an useEffect

1. **Declare an Effect.** By default, your Effect will run after every render.

```
import { useEffect } from 'react';
```

2. **Specify the Effect dependencies.** Most Effects should only re-run *when needed* rather than after every render. You can tell React to **skip unnecessarily re-running the Effect** by specifying an array of *dependencies* as the second argument to the `useEffect` call

```
useEffect(() => {
  // ...
}, []);
```

3. **Add cleanup if needed.** Some Effects need to specify how to stop, undo, or clean up whatever they were doing.

useRef

`useRef` returns a ref object with a single `current` property initially set to the initial value you provided.

On the next renders, `useRef` will return the same object. You can change its `current` property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render.

```
import { useRef } from 'react';

function Stopwatch() {
  const intervalRef = useRef(0);
  // ...
```

By using a ref, you ensure that:

- You can **store information** between re-renders (unlike regular variables, which reset on every render).
- Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).
- The **information is local** to each copy of your component (unlike the variables outside, which are shared).

```jsx
import { useState, useRef, useEffect } from 'react';

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/
      />
    </>
  );
}
```

test

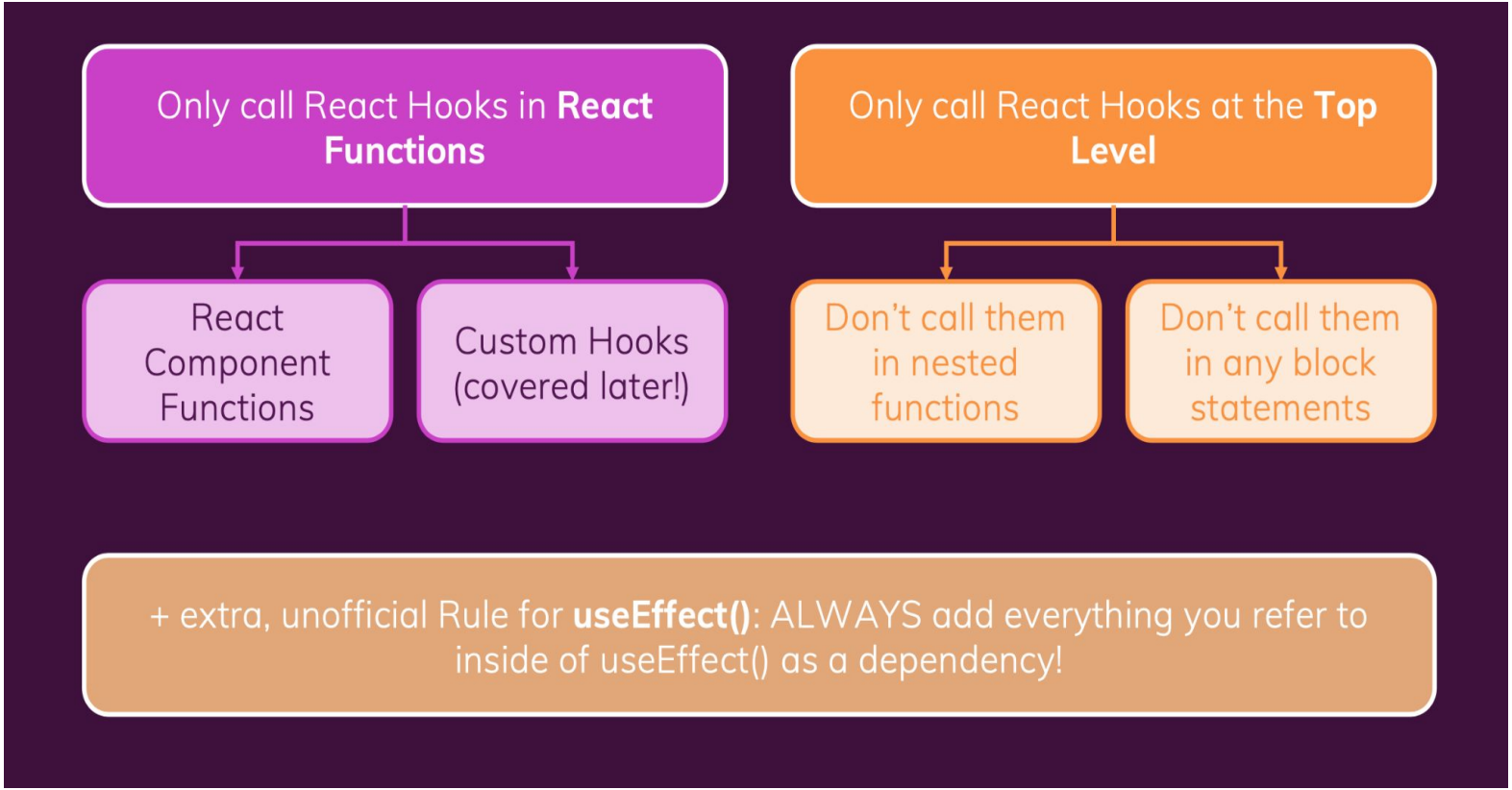
Play

CodeSandBox

Console (8)

```
Calling video.pause()
Calling video.pause()
Calling video.pause()
```

Rules of Hook



Developer Tools

/thoughtworks

React Developer tools

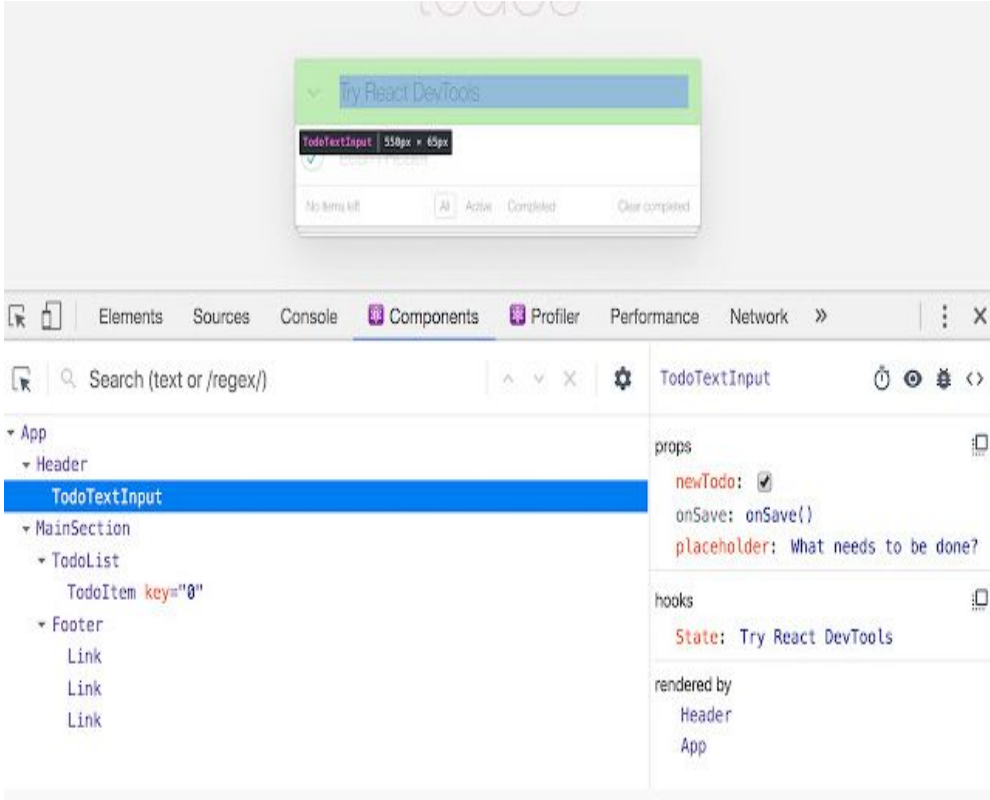
React Developer Tools is a **Chrome DevTools extension** for the open-source React JavaScript library. It allows you to inspect the React component hierarchies in the Chrome Developer Tools.

You will get two new tabs in your Chrome DevTools: "⚛ Components" and "⚛ Profiler".

The Components tab shows you the root React components that were rendered on the page, as well as the subcomponents that they ended up rendering.

By selecting one of the components in the tree, you can inspect and edit its current props and state in the panel on the right.

The Profiler tab allows you to record performance information.



Custom Hooks

/thoughtworks

Custom hooks

When we want to share logic between two JavaScript functions, we extract it to a third function. Both components and Hooks are functions, so this works for them too!

A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.

```jsx
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```jsx
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

Extracting the common logic to a custom hook

/thoughtworks

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

Using the custom hook

/thoughtworks

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

What's Next

https://roadmap.sh/react

/thoughtworks

Resources

React official Doc:

- https://reactjs.org/docs/getting-started.html
- https://beta.reactjs.org/ (in progress)

Udemy course:

- https://thoughtworks.udemy.com/course/react-the-complete-guide-incl-redux/

Other Tutorial:

- https://tkssharma.gitbook.io/react-training
- https://www.freecodecamp.org/news/react-fundamentals-for-beginners/
- https://learn.microsoft.com/en-us/training/modules/react-get-started/

Roadmap:

- https://roadmap.sh/react

Q&A

/thoughtworks