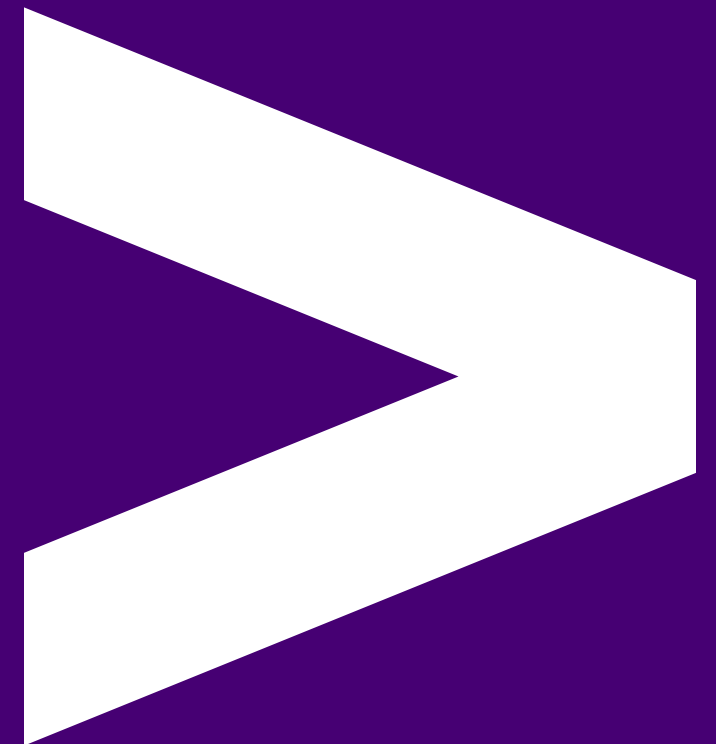


React.js 2

State



Overview

- Event-driven vs State-driven Approach
- Declarative vs Imperative Programming
- The Virtual DOM
- Unidirectional Data Flow
- Stateful vs Stateless Components
- State Management
- React Hook: useState

Objectives

- Understand the differences between a state-driven approach over an event-driven approach
- Understand what we mean by declarative or imperative programming
- Understand what the Virtual DOM is and how it works
- Understand how data and events flow in React
- Know where to put state in a React app
- Have working knowledge of the `useState` React hook

The React sessions

We have 4 React sessions:

- Components: Introduction to React, Components & JSX, Testing Components 
- State: Virtual DOM, Data Flow, State Management with the `useState` React Hook (this one)
- Side Effects: Component Lifecycle, Fetching Data with the `useEffect` React Hook, Testing User Events
- Routing: React Router, Building & Deployment



A Common Problem

How do we get elements on our page to react to user input?

I've been clicked 0 times!

Solution #1 - Event Driven



An event driven approach responds to events on the page. When the user clicks the button, we update its text directly.

- Good solution for small situations 
- Doesn't scale at all (imperative) 

[Codepen \[Event Driven\]](#)


Solution #2 - State Driven

A state driven approach responds to changes in state. When the user clicks the button, we update the state. As the button's text is mapped to a value in the state, it is indirectly updated.

- Scales beautifully (declarative) 
- Requires a paradigm shift 

[Codepen \[State Driven\]](#)

React Principles - progress

- Composition of components 
- Unidirectional Data Flow
- The Virtual DOM

React is Declarative

Declarative: Allows us to declare how our application should look and behave based on the values in our state. (React then does the leg work for us).

vs. old style...

Imperative: Requires us to directly update each and every element on our page when something happens.

The Virtual DOM

The Virtual DOM is a representation of the real DOM, but it's a lightweight copy.

It has the same properties as the real DOM, but it lacks the power to directly change what's on the page.

Manipulating the DOM is slow. Manipulating the virtual DOM is much faster, because nothing gets painted on-screen.

How it works

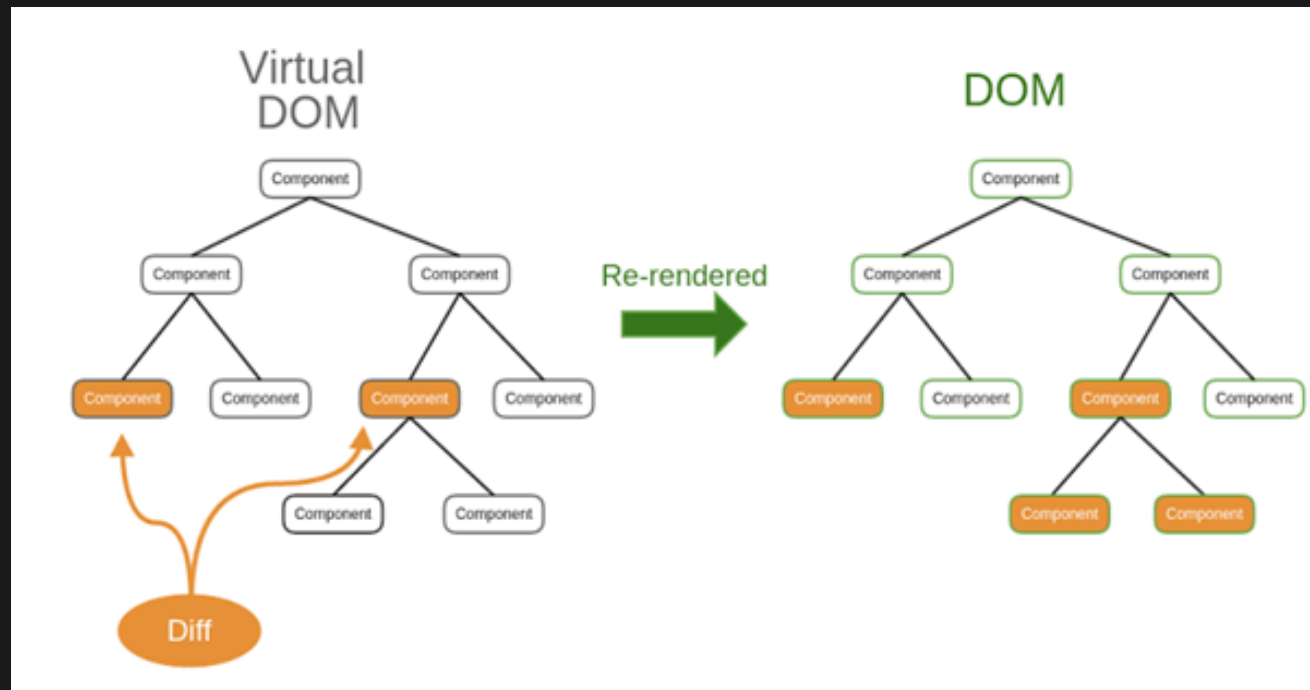
When state changes, the entire Virtual DOM is updated. This is then diffed with a previous snapshot of the Virtual DOM taken before the update occurred.

React is then able to know which real DOM components need to be updated, and updates only those, which results in a substantial performance boost.

Because of this, we should never manipulate the DOM directly. i.e:

`document.someMethod` is generally considered bad practice.

Helpful image on the following slide



Emoji Check:

How do you feel about the concept of the Virtual DOM and how it is used in React?

1. 😓 Haven't a clue, please help!
2. 😞 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 😊 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Scaling Up

Using a declarative state-driven approach allows our apps to scale with ease, provided we adhere to the principles we've covered so far.

Here is an example:

[Codepen \[Scaling Up\]](#)

What is State

State can be understood as data which represents the state of our application at any point in time. As such, any variables we define within our components become a part of our state.

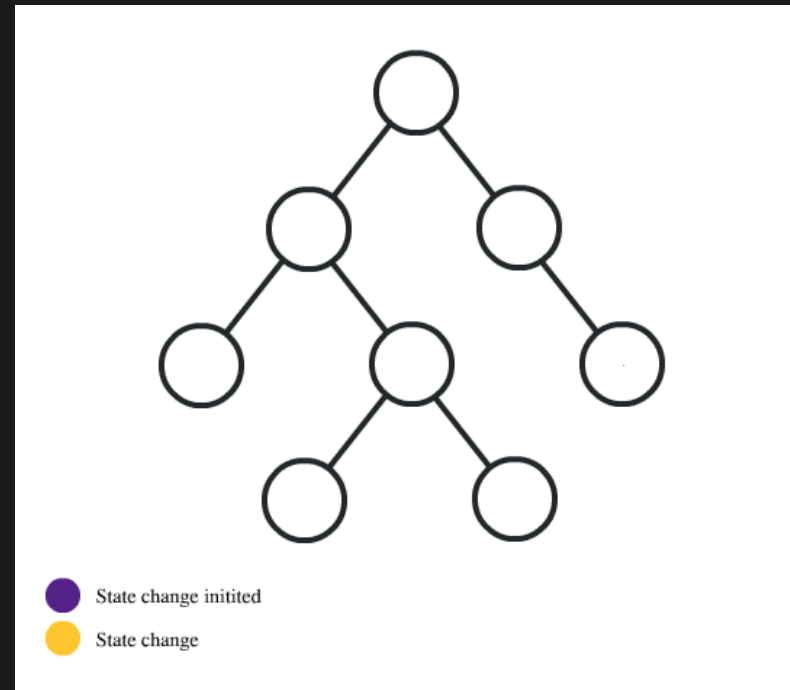
- Acts as a Single Source of Truth
- Generally considered immutable, meaning we don't mutate it directly
- State is scoped to the component which defined it
- Can become a complicated mess very quickly - good state hygiene is required

Unidirectional Data Flow

In React, data flows in one direction, from parent to child, while events bubble from child to parent.

This helps make the code less error-prone, as you have more control over your data and you know it's only flowing in one direction. It also makes testing easier as you know that changing a component's state will only ever affect its children.

Helpful gif on the following slide



State Management with Hooks

We can manage state in our React app using the included hook: `useState`
A React hook is a function that lets you "hook into" React features, in this particular case we want to add some state to our components. Hooks allow us to do that.

Hooks are pieces of reusable logic that we can share across multiple components. They help us write `DRY` code by abstracting away the complicated logic.

You can also create your own `custom hooks`.

The Rules of Hooks

We will learn about some other React hooks in future sessions, but there are some rules you should know when using them:

DO NOT

1. Call a react hook within a loop
2. Call a react hook within a conditional
3. Call a react hook within a nested function
4. Call a react hook within regular javascript functions

DO

1. Call them from within React components
2. Call them within your custom hooks

- [Rules of Hooks](#)

React Hook: useState

`useState` is a reusable function that allows us to define a state variable and a function to update it. Behind the scenes React will observe this variable and re-render the Virtual DOM for us when it changes.

`useState` accepts one argument, the initial state of our state variable. It can be of any shape.

`useState` returns an `array` containing two elements. The first is the variable holding the state, and the second is a function to update it.

Working with useState

```
// Define our state variable and update function and
// set the initial value
const [clicks, setClicks] = useState<number>(0)

console.log(clicks) // 0

// Call the provided update function to update clicks
setClicks(clicks + 1)

console.log(clicks) // 1
```

Code along - Adding state to a component - 20 mins

Now that we've had a brief introduction into how to handle state in React, let's have a go at updating a button to change colour from blue to red when you click it.

See [./exercises/react-buttons/README.md](#)

Emoji Check:

How do you feel about using the useState hook in code?

1. 😓 Haven't a clue, please help!
2. 😞 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 😊 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Stateful vs Stateless

Generally speaking, the higher up the component tree we place our state, the more components downstream we can share it with.

State then should be defined in our top-level parent components (stateful), and passed as `props` to child components (stateless).

Some child components may need to manage their own state, e.g, controlled `<input>` components, but we'll explore this later.

Codepen example of stateful vs stateless

Lets take a look at some examples of stateful and stateless components:

[Stateful components and stateless components](#)

Which do I use?

Dynamic

When dealing with dynamic information having state in the highest parent is generally the way most react projects are structured. It makes it easy to debug and track issues because all the state is managed in a single location, instead of it being shared amongst the components.

Static

If the information in a component will always be static (like a page description) and is not needed by other components then it is ok to have the state managed within that component.

Managing State: Gotchas

Lets take a look at the following code pen problem:

[Component state problem](#)

Explanation on next slide

useState considerations

The example on the last slide shows a complex/non-trivial state variable, but the handlers don't update all of it, only one key each.

But of course, react re-renders every affected component.

So we must provide a full copy of the state with only the required bits changed.

Merging Previous State

`setState` will replace the entire `state` variable, so when state is an object / array and we wish to only update one entry, we need to make a new object / array and merge in our changes:

```
type State = {  
  colour: 'red' | 'blue'  
  clicks: number  
}  
  
const [currentState, setState] = useState<State>({  
  colour: "blue",  
  clicks: 0,  
})  
// later...  
const clickHandler () => {  
  const newState = {...currentState}  
  newState.clicks = currentState.clicks + 1  
  setState(newState)  
}
```

Managing state separately

In react its best practice however to manage the state separately, especially if there is no logical connection between the values. Here is another way to manage this example:

```
const [clicks, setClicks] = useState<number>(0)
const [colour, setColour] = useState<'blue' | 'red'>("blue")

setClick(clicks + 1)
setColour("red")
```

This lets separate functions manage the state it cares about preventing you from having to track the other values.

Emoji Check:

How do you feel about the concept of state and how to use it in a React app?

1. 😓 Haven't a clue, please help!
2. 😞 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 😊 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Handling User Input

Suppose we added an `<input>` element to get the user's name. Traditionally we'd have something imperative like this that accesses the DOM directly:

```
const name = document.getElementById("name").value
```

In React we should not be accessing or manipulating the DOM directly as this bypasses the Virtual DOM.

Handling User Input in React

In React we do not want our state stored in a HTML element. We want to be able to answer the question "what is the value of the input right now" without having to reach into the DOM to find out.

So we should use the following controlled input:

```
const [name, setName] = useState<string>('')  
  
<input  
  value={name}  
  onInput={(event) => setName(event.target.value)}/>
```

Controlled vs Uncontrolled inputs

Controlled inputs are where, as the previous slide, we record the state at all times. This is the react way.

Uncontrolled inputs are where we ignore this best practice and only look at it's value later. This is uncommon but there may be valid reasons in some use cases, so we would be remiss not to mention it here.

Why do we prefer controlled components?

With controlled components we can answer the question "what is the value of the input right now" via the state.

We want to centralise all our state info (the source of truth) in our React app.

With uncontrolled components the state of the input is stored in the HTML, and we have to reach into the DOM to answer that question - and in React we should not be poking about in the DOM!

...An example where this might be useful is when having to integrate React and non-React code. Or managing focus, text selection, or media playback.

...but you should usually use controlled components!

Controlled Component Demo

Every time you input a character into the input box the state is updated, which triggers a re-render of the component.

[Codepen \[How many renders was that?\]](#)

As you type your input gets stored in the state and then rendered from the state rather than from your input.

Uncontrolled Components in React

Sometimes there may be valid reasons to not use a controlled component. Of course, React have provided a way to do this so that we at least don't break the core rule of "never use `document.xyz` methods"

In this case we can leverage the `useRef` react hook. This provides a reference to the element we want to track via the ref's `.current` property:

```
// import...
const inputElement = useRef<HTMLInputElement | null>(null)

// attach to the element...

```

Uncontrolled Components Example

Assuming `onSave()` is a utility function defined elsewhere:

```
const inputElement = useRef<HTMLInputElement | null>(null)

const handleClick = () => {
  onSave(inputElement.current.value) // for example
}

return (
  <div>
    <input type="text" ref={inputElement} />
    <button onClick={handleClick}>Save</button>
  </div>
)
```

Controlled vs Uncontrolled Components Demo

An example of Controlled and Uncontrolled components:

[Codepen \[Controlled vs Uncontrolled\]](#)

With an uncontrolled input you have to 'pull' the value from the field when you need it, typically this will be only once - when the form is submitted.

With a controlled input we know what the current value of the input is at all times.

A useful article about Controlled and Uncontrolled components: [Controlled vs Uncontrolled](#)

Emoji Check:

How you feel about the concept of controlled inputs in React?

1. 🥲 Haven't a clue, please help!
2. 😞 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 😊 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Quiz time!

A chess app built in React needs to track the current layout of the board for many functions. How should this state be stored?

- A. In `props`
- B. Using `useState()` at the top of the component tree
- C. In a global variable
- D. Using `useState()` as close to the board component as possible

Answer

The Answer is B.

We want to use this state in multiple functions, so we want its state to be fully handled at the highest part of the component tree. This ensures that whenever a function changes it, it will refresh the entire DOM with that change preventing any state issues.

Quiz time!

What is the correct syntax for React's useState hook?

- A. `let [score, setScore] = useState(0);`
- B. `const [score, setScore] = useState(0);`
- C. `const [score, setScore] = useState();`
- D. `const score = useState();`

Answer

The answer is B!

We use a const because we want to avoid accidentally mutating the state variable `score` returned to us from `useState`.

We also need a starting state to be passed into the function so we have an initial value to work with.

Quiz time!

What React feature allows it to efficiently re-render the entire app multiple times per second in memory, but only the bits of the web page that really need it?

- A. Hooks
- B. Components
- C. Virtual DOM
- D. Transpilation

Answer

The answer is C!

The Virtual DOM allows us to efficiently detect changes to any component that's been updated, and then update the DOM with them instantaneously.

Quiz time!

Which of these places would it be ok to use a React hook?

- A. Inside an `if/else` block
- B. Inside a `for` loop
- C. Inside a regular js function
- D. None of the above

Answer

The answer is D.

The other answers all break the React hook rules:

- The first two (A, B) would create an unpredictable path for the hooks to run in as some may or may not happen based on the conditions.
- And (C) the regular JS function would be taking stateful logic outside of the component.

Quiz time!

Which of these statements is correct about a React app?

- A. Events flow from parent to child, data bubbles from child to parent
- B. Data flows from parent to child, events bubble from child to parent
- C. Events and Data flow from parent to child
- D. Events and Data bubble from child to parent

Answer

The answer is B!

The parents pass data down to the children so that can display it, and events that occur within the children then bubble up to parent where it can act on those. This will then update its children if the state has changed.

Exercise - Todo App - continued - 30mins

Instructor to distribute exercise:

See [./exercises/react-todo/README.md](#)

Overview – recap

- Event-driven vs State-driven Approach
- Declarative vs Imperative Programming
- The Virtual DOM
- Unidirectional Data Flow
- Stateful vs Stateless Components
- State Management
- React Hook: `useState`

Objectives - recap

- Understand the differences between a state-driven approach over an event-driven approach
- Understand what we mean by declarative or imperative programming
- Understand what the Virtual DOM is and how it works
- Understand how data and events flow in React
- Know where to put state in a React app
- Have working knowledge of the `useState` React hook

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😓 Haven't a clue, please help!
2. 😞 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 😊 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively