React 3: State & Lifecycle Methods

IN608: Intermediate Application Development Concepts

Kaiako: Tom Clark & Grayson Orr

Last Session’s Content

- Components
  - Function
  - Class
- Props

Today’s Content

- State
- Lifecycle methods
  - Mounting
  - Updating
  - Unmounting
- React Hooks
  - useState
  - useEffect
- Data flow

State

State

- Consider the following component:

```
import React from 'react'

class Owner extends React.Component {
  render() {
    return <h1>My owner is {this.props.name}</h1>
  }
}

export default Owner
```

- We are going to add local state to this class component

State

- In the `render()`, replace `this.props.name` with `this.state.name`
- Add a class constructor which assigns the initial `this.state`
- What is state?
  - Contains data specific to the component
  - Data may change over time
  - State is user-defined & should be a JavaScript object
  - Never mutate `this.state` directly, i.e., `this.state.name = 'John Doe'`. Instead, use `setState()`
- All class components should always call the base constructor with `props`

```
import React from 'react'

class Owner extends React.Component {
  constructor(props) {
    super(props)
    this.state = { name: 'Jane Doe' }
  }

  render() {
    return <h1>My owner is {this.state.name}</h1>
  }
}

export default Owner
```

State

- Create a new component file called `Clock.js`
- Later, we are going to add lifecycle methods to this class component

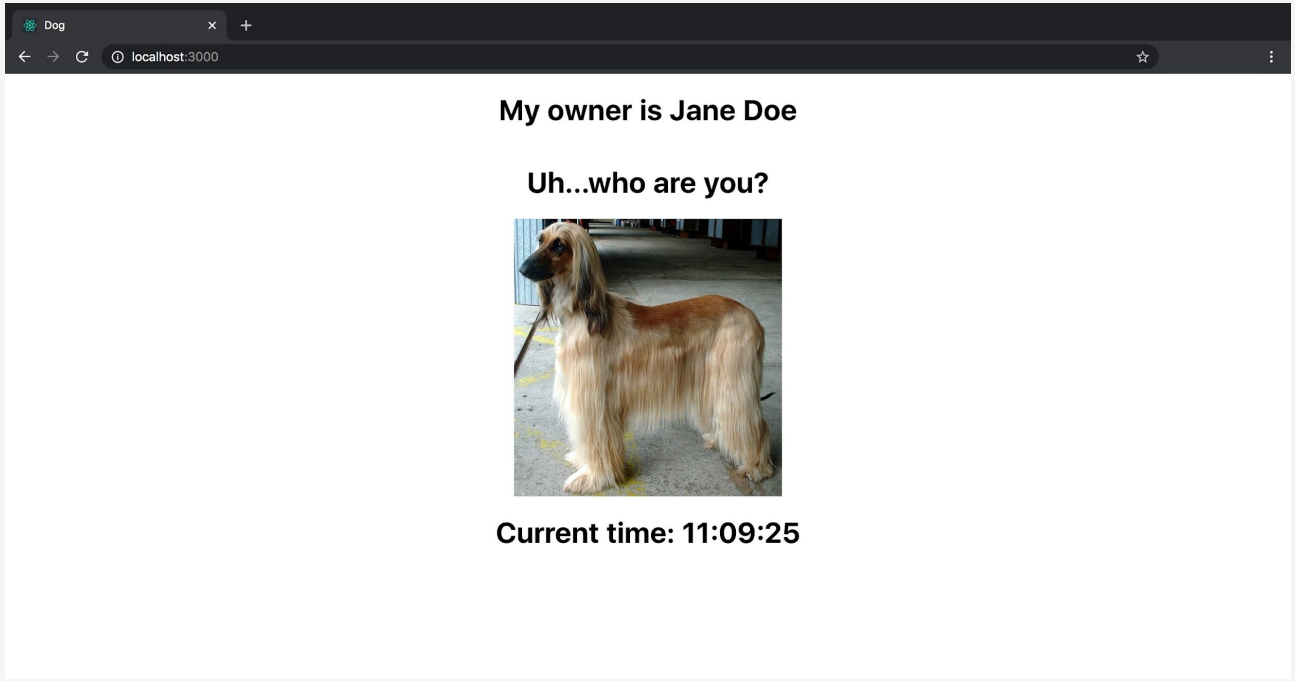
```
import React from 'react'

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

  render() {
    return <h1>Current time: {this.state.date.toLocaleTimeString()}</h1>
  }
}

export default Clock
```

State

- In `App.js`

```
import React from 'react'
import Clock from './Clock'
import Owner from './Owner'
import afghanHoundImg from '../img/afghan-hound.jpg'

const App = () => {
  const dog = {
    name: 'Bingo',
    breed: 'Afghan Hound',
    img: afghanHoundImg,
  }

  const formatDog = (dog) =>
    `Woof woof, my name is ${dog.name} & my breed is an ${dog.breed}`

  const getGreeting = (dog) => {
    if (dog) {
      return <h1>{formatDog(dog)}</h1>
    }
    return <h1>Uh...who are you?</h1>
  }

  return (
    <div className='main-container'>
      <Owner />
      {getGreeting()}
      <img src={dog.img} alt='afghan hound' width='300' />
      <Clock />
    </div>
  )
}

export default App
```

State


Dog
localhost:3000
My owner is Jane Doe
Uh...who are you?
Current time: 11:09:25

Lifecycle Methods

Lifecycle Methods

- Each component has several lifecycle methods which can be overridden to run at specific times during the process
- The component lifecycle is broken up into the following:
  - Mounting
  - Updating
  - Unmounting
  - Error handling
- We will only be concerned with the first three
- The mostly commonly used methods are in **bold**

Mounting

- When a component is being created & inserted into the DOM, the following methods are called:
  - `constructor()`
    - Called before the component is mounted
    - When implementing, `super(props)` should always be called first
    - May lead to bugs because `this.props` will be undefined
    - If you do not initialise state or do not bind methods, you do not have to implement `constructor()`
  - `static getDerivedStateFromProps()`
  - `render()`
    - The only required method for a class component
    - Does not modify component state
    - When invoked, returns the same result each time
  - `componentDidMount()`
    - Invoked immediately after a component is mount, i.e., inserted into the DOM tree

Updating

- When changes are made to props or state, the following methods are called:
  - `static getDerivedStateFromProps()`
  - `shouldComponentUpdate()`
  - **`render()`**
  - `getSnapshotBeforeUpdate()`
  - **`componentDidUpdate()`**
    - Invoked immediately after an update occurs
    - This is not called for the initial render when mounting a component

Unmounting

- When a component is removed from the DOM, the following method is called:
  - `componentWillMount()`
    - Invoked immediately before a component is unmounted & destroyed
    - Necessary cleanup is performed, i.e, cancelling network requests

Lifecycle Methods

- Declare a method called `tick()`
- `this.setState()`
  - Enqueues changes to the component
  - Tells React that this component, i.e., `Clock` & its children need to be re-rendered with the update state, i.e., `new Date()`
  - A component does not always immediately update. It may batch or defer the update until later

```
import React from 'react'

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

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

  render() {
    return <h1>Current time: {this.state.date.toLocaleTimeString()}</h1>
  }
}

export default Clock
```

Lifecycle Methods

- Declare `componentDidMount()` & `componentWillUnmount()` lifecycle methods

```
import React from 'react'

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

  componentDidMount() {}

  componentWillUnmount() {}

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

  render() {
    return <h1>Current time: {this.state.date.toLocaleTimeString()}</h1>
  }
}

export default Clock
```

Lifecycle Methods

- You can add additional fields to a class component, i.e., `this.timerID`
- `setInterval(callback, delay)` - schedules repeated executions of a `callback`, i.e., `tick()` every `delay` milliseconds. Returns a `Timeout` object

```
import React from 'react'

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

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

  componentWillUnmount() {}

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

  render() {
    return <h1>Current time: {this.state.date.toLocaleTimeString()}</h1>
  }
}

export default Clock
```

Lifecycle Methods

- `clearInterval(timeout)` - cancels a `Timeout` object created by `setInterval()`

```
import React from 'react'

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 <h1>Current time: {this.state.date.toLocaleTimeString()}</h1>
  }
}

export default Clock
```

Lifecycle Methods

- Lets recap...
  - When `<Clock />` is passed to `ReactDOM.render()`, React calls the `Clock` component's constructor
  - Initialises local state by assigning an object to `this.setState()`
  - React calls the `Clock` component's `render()` method
  - React updates the DOM to match the `Clock` component's render output
  - When the output is inserted into the DOM, React calls the `componentDidMount()` lifecycle method
  - `Clock` component asks the browser to setup a timer which calls the `tick()` method every 1000 milliseconds
  - React knows the state has changed & calls the `render()` method again
  - If the `Clock` component is removed from the DOM, React calls the `componentWillUnmount()` lifecycle method

React Hooks

React Hooks

- What is a Hook?
  - Functions that let you “hook into” React state & lifecycle methods from a function component
  - Hooks do not work inside class components - they let you use React without classes
- When to use a Hook?
  - If you write a function component & you need to add some state to it
- Resource: https://reactjs.org/docs/hooks-overview.html

React Hooks - useState

- In a class component, we initialise `date` to `new Date()` by setting `this.state` to `{ date: new Date() }` in the constructor

```
import React from 'react'

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

  ...
```

- In a function component, we have no `this`, meaning we can not assign or read `this.state`
- Instead, we call the `useState` Hook directly in our component

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

const Clock = () => {
  const [date, setDate] = useState(new Date())

  ...
```

React Hooks - useState

- What does calling `useState` do?
  - It declares a state variable
  - A new way to use the exact same capabilities that `this.state` provides in a class component
  - This is a way to preserve values between function calls
  - Normally, variables disappear when the function exits, but state variables are preserved by React
- What do we pass to `useState` as an argument?
  - The only argument to the `useState()` Hook is the initial state
  - The state does not have to be an object - it can be a boolean, number, string, null, etc
  - If we wanted to store two different values in state, we would have to call `useState()` twice
- What does `useState` return?
  - It returns a pair of values - the current state & a function that updates it, i.e., `date` & `setDate`
  - This is similar to `this.state.date` & `this.setState()` in a class component, but in a function component, you get them in a pair
- Resource: https://reactjs.org/docs/hooks-state.html

React Hooks - useState

- Consider the following example:

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

const Clock = () => {
  const [date, setDate] = useState(new Date())

  // Updating state
  const tick = () => setDate(new Date())

  // Reading state
  return <h1>Current time: {date.toLocaleTimeString()}</h1>
}

export default Clock
```

React Hooks - useEffect

- Data fetching, setting up a subscription & manually changing the DOM in React components are all example of side effects or effects

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

const Clock = () => {
  const [date, setDate] = useState(new Date())

  const tick = () => setDate(new Date())

  useEffect(() => {
    const timerID = setInterval(() => tick(), 1000)
    return () => clearInterval(timerID) // Cleanup method
  }, [])

  return <h1>Current time: {date.toLocaleTimeString()}</h1>
}

export default Clock
```

- You can think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate` & `componentWillUnmount` combined

React Hooks - useEffect

- What does `useEffect` do?
  - Tells React that your component needs to do something after it renders
  - React remembers the function/effect you passed in & calls it later after performing the DOM updates
- Why is `useEffect` called inside a component?
  - Lets you access state variables & props right from the effect
  - We do not need an API to access them as it is already in the function scope
  - Hooks uses JavaScript closures & avoids React-specific APIs
- Does `useEffect` run after every render?
  - It runs both after the first render & after every update
  - You might find it easier to think that effects happen “after render” rather than “mounting” & “updating”
- Resource: https://reactjs.org/docs/hooks-effect.html

Data Flow

Data Flow

- Parent & child component can know if a component is stateful or stateless
- Components should not care if it defined as a function or class
- State is often called local or encapsulated...what does this mean? It is not accessible to any component other than the one that sets it
- A component may choose to pass its state to its child components as props, i.e., a child component would receive `date` in its props & would not know whether is came from the `Clock` state or `Clock` props
- This type of data flow is top-down meaning state is always owned by a specific component & data from that state can only affect components below them in the DOM tree

Programming Activity

- Checkout to master - `git checkout master`
- Create a new branch called 18-practical - `git checkout -b 18-practical`
- Open the file `18-practical.pdf` and work on the tasks described