MOBA2

MOBILE WEB: REACT

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

PROPERTIES AND STATE

- Component data comes in two varieties
- State is the dynamic part of a React component
- Properties are used to pass data into components

Whenever we tell a React component to change its state, the component will automatically re-render itself

INITIAL COMPONENT STATE

```
class MyComponent extends Component {

  state = {                  // The initial state is set as a simple
    first: false,            // property of the component instance -
    second: true,            // it should always be an object
  }

  render () {
    const { first, second } = this.state

    return (
      <main>
        <section>
          <button disabled={first}>First</button>
        </section>
        <section>
          <button disabled={second}>Second</button>
        </section>
      </main> )
  }
}
```

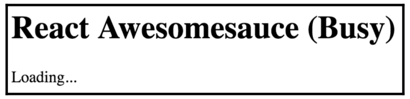
SETTING COMPONENT STATE (1)

```
class MyComponent extends Component {

  state = {
    heading: 'React Awesomesauce (Busy)',
    content: 'Loading...',
  }

  render () {
    const { heading, content } = this.state
    return (
      <main>
        <h1>{heading}</h1>
        <p>{content}</p>
      </main>
    )
  }
}
```

SETTING COMPONENT STATE (2)

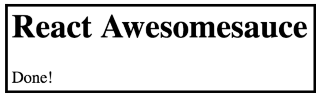
```
// The "render()" function returns a reference to the
// rendered component. In this case, it's an instance
// of "MyComponent". Now that we have the reference,
// we can call "setState()" on it whenever we want.
const myComponent = render(
  (<MyComponent />),
  document.getElementById('app')
)

// After 3 seconds, set the state of "myComponent",
// which causes it to re-render itself.
setTimeout(() => {
  myComponent.setState({
    heading: 'React Awesomesauce',
    content: 'Done!',
  })
}, 3000)
```

SETTING COMPONENT STATE



React Awesomesauce (Busy)

Loading...



React Awesomesauce

Done!

MERGING COMPONENT STATE

- Calling `setState()` doesn't replace the state
- The object that you pass is *merged* to the state
- You can set individual state properties on components

PASSING PROPERTY VALUES

- Properties get passed into components
- They're only set once, when the component is rendered
- We can pass just about anything as a property value via JSX
- As long as it's a valid JavaScript expression
- Properties are available in the component as `this.props`

PASSING PROPERTY VALUES (1)

```
const appState = {
  text: 'My Button',
  disabled: true,
}

render((
  <main>
    <MyButton
      text={appState.text}
      disabled={appState.disabled}
    />
  </main>
  ),
  document.getElementById('app')
)
```

PASSING PROPERTY VALUES (2)

```
class MyButton extends Component {

  render() {
    const { disabled, text } = this.props

    return (
      <button disabled={disabled}>{text}</button>
    )
  }
}
```

- `this.props`: property values passed to component
- `this.props.children`: child elements of component

DEFAULT PROPERTY VALUES

```
class MyButton extends Component {

  // The "defaultProps" values are used when the
  // same property isn't passed to the JSX element.
  static defaultProps = {
    disabled: false,
    text: 'My Button',
  }

  render () {
    const { disabled, text } = this.props

    return (
      <button disabled={disabled}>{text}</button>
    )
  }
}
```

FUNCTION COMPONENTS

```
// Class-based React component
class MyButton extends Component {
  render () {
    const { disabled, text } = this.props

    return (
      <button disabled={disabled}>{text}</button>
    )
  }
}
```

```
// Function component
const MyButton = ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
)
```

FUNCTION COMPONENTS

- Previously, often called *Stateless Functional Components*
- It's just what it sounds like – a function
- Given some properties, it returns the component's JSX
- React Hooks allow function components with state and lifecycle

DEFAULTS IN FUNCTION COMPONENTS

```
// Object destructuring with defaults
const MyButton = ({ disabled=false, text='My Button'}) => (
  <button disabled="{disabled}">{text}</button>
)
```

REACT HOOKS

- New addition in React 16.8 (and React Native 0.59)
- Use state and other React features without writing a class
- Completely opt-in and 100% backwards-compatible
- No plans to remove classes from React
- More direct API to React concepts: props, state, context, refs, and lifecycle

EXAMPLE: STATE HOOK

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

function Example () {

  const [count, setCount] = useState(0)

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

WHAT IS A HOOK?

- Functions that let you “hook into” React state and lifecycle
- Hooks let you use React without classes
- There are a few built-in Hooks like `useState`

MULTIPLE STATE VARIABLES

```
const ExampleWithManyStates = () => {

  // Declare multiple state variables!
  const [age, setAge] = useState(42)
  const [fruit, setFruit] = useState('banana')
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }])

  // ...
}
```

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

DECLARING HANDLER FUNCTIONS

```
function MyButton = (props) => {

  const handleClick = () => {
    console.log('clicked')
  }

  // Renders a "<button>" element with the "onClick" event handler
  // set to the "handleClick()" function.
  return (
    <button onClick={handleClick}>
      {props.children}
    </button>
  )
}
```

HANDLER IN AN CLASS COMPONENT

```
class MyButton extends Component {

  handleClick () {
    console.log('clicked')
  }

  // Renders a "<button>" element with the "onClick" event handler
  // set to the "handleClick()" method of this component.
  render () {
    return (
      <button onClick={this.handleClick}>
        {this.props.children}
      </button>
    )
  }
}
```

DECLARING HANDLER FUNCTIONS

- Event handlers for particular elements are declared in JSX
- Elements can have more than one event handler
- List of supported events: https://reactjs.org/docs/events.html

EVENT HANDLER CONTEXT

- Event handlers usually need access to properties or state
- In React, they don't pull data out of DOM elements
- Methods must be manually bound to the component context

```
<button onClick={handleclick.bind(this)}>Start</button>
```

```
// or:
constructor () {
  super()
  this.handleclick = this.handleclick.bind(this)
}
return (
  <button onClick={handleclick}>Start</button>
)
```

INLINE EVENT HANDLERS

```
class MyButton extends Component {

  // Renders a button element with an "onClick()" handler.
  // This function is declared inline with the JSX, and is
  // useful in scenarios where you need to call another
  // function.
  render () {
    return (
      <button
        onClick={e => console.log('clicked', e)}
      >
        {this.props.children}
      </button>
    )
  }
}
```

BINDING HANDLERS TO ELEMENTS

- React doesn't attach event listeners to the DOM elements
- Handlers are added to an internal mapping
- There's a single event listener on the root DOM container into which the React tree is rendered
- React < v17.0 : event listener was on the document node

EVENT OBJECT

- Event handler will get an event argument passed to it
- This event object is a wrapper for native event instances
- It is sometimes known as a synthetic event

React event object (beta docs)

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

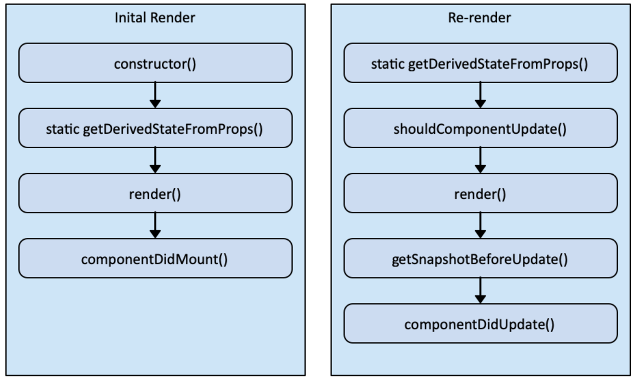
COMPONENT LIFECYCLE

React components go through a lifecycle

- Component is about to be mounted
- Component is rendered
- After the component has been mounted
- When the component is updated

... and so on

CLASS COMPONENT LIFECYCLE



SIMULATE API ACCESS

```
function users(fail) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (fail) {
        reject('epic fail')
      } else {
        resolve({
          users: [
            { id: 0, name: 'First' },
            { id: 1, name: 'Second' },
            { id: 2, name: 'Third' },
          ],
        })
      }
    }, 2000)
  })
}
```

FETCHING DATA

```
class UserListContainer extends Component {
  state = {
    data: {
      error: null,
      loading: 'loading...',
      users: [],
    },
  }

  componentDidMount() {...}

  render () {
    return ( <UserList {...this.state.data} /> )
  }
}
```

LIFECYCLE METHOD

```
componentDidMount() {
  users().then(      // users(true) to reject Promise
    (result) => {
      this.setState({
        data: {
          error: null,
          loading: null,
          users: result.users,
        },
      })
    },
    (error) => {
      this.setState({
        data: {
          error: error,
          loading: null,
          users: this.state.users,
```

UI COMPOMENTS

```
const UserList = ({ error, loading, users }) => (
  <section>
    <ErrorMessage error={error} />
    <LoadingMessage loading={loading} />
    <ul>
      {users.map(i => (
        <li key={i.id}>{i.name}</li>
      ))}
    </ul>
  </section>
)
```

```
const ErrorMessage = ({ error }) =>
  error ? (<strong>{error}</strong>) : null
```

```
const LoadingMessage = ({ loading }) =>
  loading ? (<strong>{loading}</strong>) : null
```

OPTIMIZE RENDERING EFFICIENCY

- If the state hasn't changed, there's no need to render
- If the `shouldComponentUpdate()` method returns `false`, no render happens
- Useful if the component is rendering a lot of data and is re-rendered frequently

THE EFFECT HOOK

- Tell React what to do after render
- Argument is a function (the effect)
- Function will be called after performing the DOM updates
- It can use the state variables (closure)

Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution

THE EFFECT HOOK

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

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    document.title = `You clicked ${count} times`
  })

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

EFFECTS WITH CLEANUP

- Many effects don't require any cleanup when the component unmounts, but some effects do
- Example: Component subscribes to some external data source
- In a class-based component: lifeycle method `componentWillUnmount`
- With Hooks: effect returns a cleanup function

EFFECTS WITH CLEANUP

```
function FriendStatus (props) {
  const [isOnline, setIsOnline] = useState(null)

  useEffect(() => {
    function handleStatusChange (status) {
      setIsOnline(status.isOnline)
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange)
    // Specify how to clean up after this effect:
    return function cleanup () {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange)
    }
  })

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

EFFECT HOOK PERFORMANCE

- Cleanup is performed when the component unmounts
- However, effects run for every render
- React also cleans up effects from the previous render
- We can skip applying an effect if certain values haven't changed
- Pass an array of these variables as an optional second argument to `useEffect`

EFFECT HOOK PERFORMANCE

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

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange)
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange)
  }
}, [props.friend.id])  // Only re-subscribe if props.friend.id changes
```

EFFECT HOOK PERFORMANCE

```
// Passing a dependency array
useEffect(() => {
  // ...
}, [a, b])   // Runs again if a or b are different

// Passing an empty dependency array
useEffect(() => {
  // ...
}, [])       // Does not run again

// Passing no dependency array at all
useEffect(() => {
  // ...
})           // Always runs again
```

RULES OF HOOKS

- Only call Hooks at the top level
  - don't call Hooks inside loops, conditions, or nested functions
  - ensure that Hooks are called in the same order each time a component renders
- Only call Hooks from React functions
  - call Hooks from React function components
  - or call Hooks from custom Hooks

BUILDING CUSTOM HOOKS

- Extract component logic into reusable functions
- A custom Hook is a function that may call other Hooks
- A custom Hook's name starts with use

BUILDING CUSTOM HOOKS

```
import React, { 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
}
```

BUILDING CUSTOM HOOKS

- Friend status logic can now bee removed from components
- The custom Hook *useFriendStatus* is used instead

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

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

BUILDING CUSTOM HOOKS

- Advantage: it can be used in other components, too
- All state and effects inside a custom Hook are fully isolated

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

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

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

CONTAINER COMPONENTS

- Common React pattern: concept of container components
- Don’t couple data fetching with data rendering
- The container is responsible for fetching the data
- Data is then passed down to a component responsible for rendering the data

CONTAINER COMPONENTS (1)

```
// Utility function that’s intended to mock a service that this
// component uses to fetch it's data. It returns a promise, just
// like a real async API call would. In this case, the data is
// resolved after a 2 second delay.

function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve([ 'First', 'Second', 'Third' ])
    }, 2000)
  })
}
```

CONTAINER COMPONENTS (2)

```
const MyContainer = () => {

  const [items, setItems] = useState([])

  useEffect(() => {
    fetchData()
      .then(items => setItems(items))
  }, [])       // run on first render

  return (
    <MyList {...this.state} />
  )
}
```

CONTAINER COMPONENTS (3)

```
// A stateless function component that expects an "items"
// property so that it can render a "<ul>" element.

const MyList = ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    ))}
  </ul>
)
```

CONTEXT API

- Container components fetch and manipulate data
- Data is passed down to components for rendering
- Typically, data is passed top-down via props
- This can be cumbersome for certain types of props
- Examples: locale preferences, UI theme
- Data that can be considered “global” for a tree of components
- Context provides a way to share values between components

EXAMPLE WITHOUT CONTEXT API

```
class App extends Component {
  render () {
    return <Toolbar theme="dark" />
  }
}
function Toolbar(props) {
 return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  )
}
function ThemedButton(props) {
  return (
    <Button theme={props.theme} />
  )
}
```

EXAMPLE WITH THE CONTEXT API

```
const themes = {
  light: { foreground: "#000000", background: "#eeeeee" },
  dark:  { foreground: "#ffffff", background: "#222222" },
}

const ThemeContext = React.createContext(themes.light)

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  )
}
```

USING THE CONTEXT HOOK

```
function Toolbar (props) {   // no need to pass down the theme
  return (
    <div>
      <ThemedButton />
    </div>
  )
}

function ThemedButton () {
  const theme = useContext(ThemeContext)

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  )
}
```

CONTEXT HOOK

- `useContext` accepts a context object
- It returns the current context value for that context
- You still need a `<MyContext.Provider>` above in the tree
- When the nearest context provider updates, the Context Hook triggers a re-render of the component
- If re-rendering is expensive, you can use memoization

OVERVIEW

- Properties and State
- Event Handling
- Component Lifecycle
- Container Components
- Developer Tools

INSTALLATION

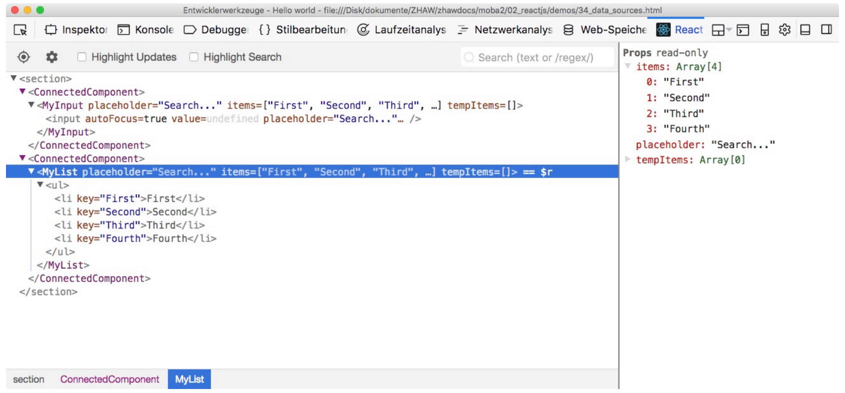
```
$ npx create-react-app hello-world
$ cd hello-world/

$ npm start
Starting the development server...
Compiled successfully!
The app is running at: http://localhost:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

- cf. React toolchains
- npx is a npm package runner

DEVELOPMENT ENVIRONMENT

- Install *React Devtools* in your browser (Firefox, Chromium)
  https://github.com/facebook/react/tree/master/packages/react-devtools
  Allows inspection of React component hierarchy
- Install JSX support in your editor
  - VSCode: Basic support available out-of-the box
  - The JavaScript language extension provides additional features

REACT DEVTOOLS



```
▼<section>
  ▼<ConnectedComponent>
    ▼<MyInput placeholder="Search..." items=["First", "Second", "Third", …] tempItems=[]>
       <input autoFocus=true value=undefined placeholder="Search..."… />
     </MyInput>
   </ConnectedComponent>
  ▼<ConnectedComponent>
    ▼<MyList placeholder="Search..." items=["First", "Second", "Third", …] tempItems=[]> == $r
      ▼<ul>
         <li key="First">First</li>
         <li key="Second">Second</li>
         <li key="Third">Third</li>
         <li key="Fourth">Fourth</li>
       </ul>
     </MyList>
   </ConnectedComponent>
 </section>
```

```
Props read-only
▼ items: Array[4]
    0: "First"
    1: "Second"
    2: "Third"
    3: "Fourth"
  placeholder: "Search..."
▶ tempItems: Array[0]
```

VALIDATING COMPONENT PROPERTIES

- Goal: knowing what’s passed into the component
- Validation emits a warning when something doesn’t pass
- In production mode, property validation is turned off

https://www.npmjs.com/package/prop-types

READING MATERIAL, SOURCES

SOURCES

- React – A JavaScript library for building user interfaces
  https://reactjs.org
- Adam Boduch: React and React Native
  Second Edition, Packt Publishing, 2018
  Packt Online Shop