

**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

```
1 import React, { useState } from 'react'
2
3 function Example () {
4
5     const [count, setCount] = useState(0)
6
7     return (
8         <div>
9             <p>You clicked {count} times</p>
10            <button onClick={() => setCount(count + 1)}>
11                Click me
12            </button>
13        </div>
14    )
15 }
```

# 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

```
1 function MyButton = (props) => {  
2  
3   const handleClick = () => {  
4     console.log('clicked')  
5   }  
6  
7   // Renders a "<button>" element with the "onClick" event handler  
8   // set to the "handleClick()" function.  
9   return (  
10     <button onClick={handleClick}>  
11       {props.children}  
12     </button>  
13   )  
14 }
```

# HANDLER IN AN CLASS COMPONENT

```
1  class MyButton extends Component {  
2  
3    handleClick () {  
4      console.log('clicked')  
5    }  
6  
7    // Renders a "<button>" element with the "onClick" event handler  
8    // set to the "handleClick()" method of this component.  
9    render () {  
10     return (  
11       <button onClick={this.handleClick}>  
12         {this.props.children}  
13       </button>  
14     )  
15   }  
16 }
```

# 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

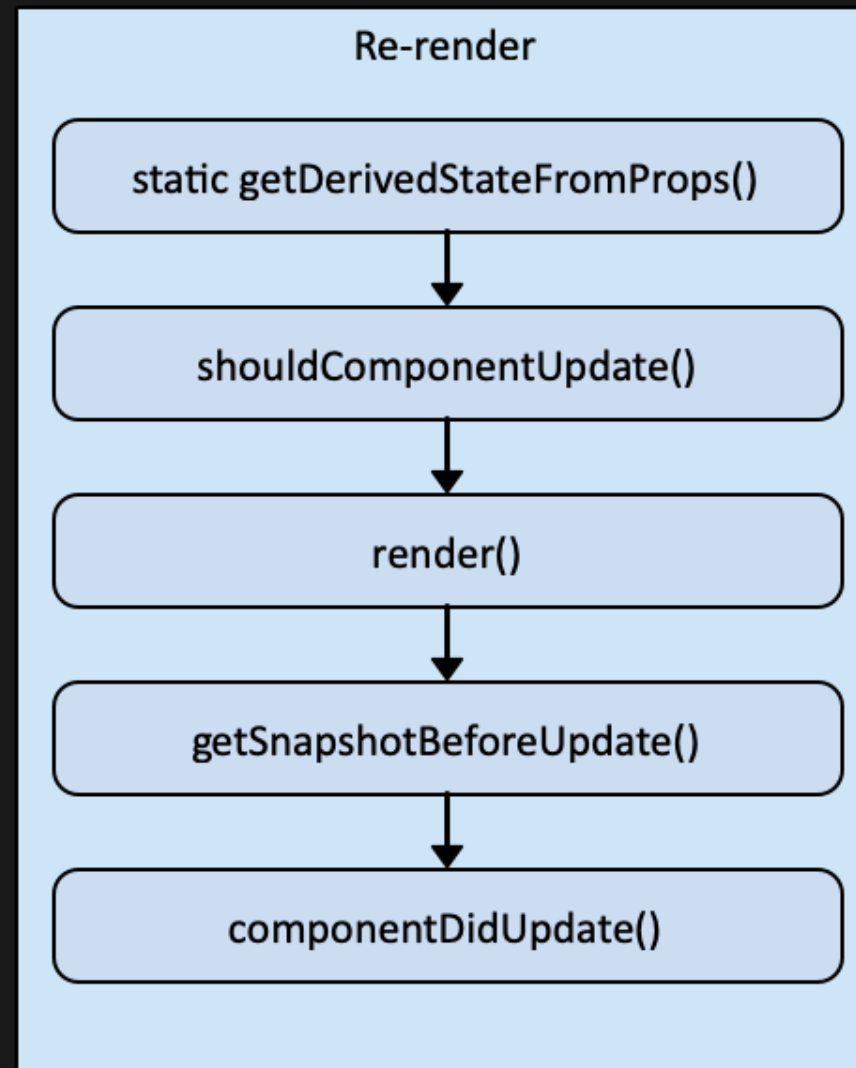
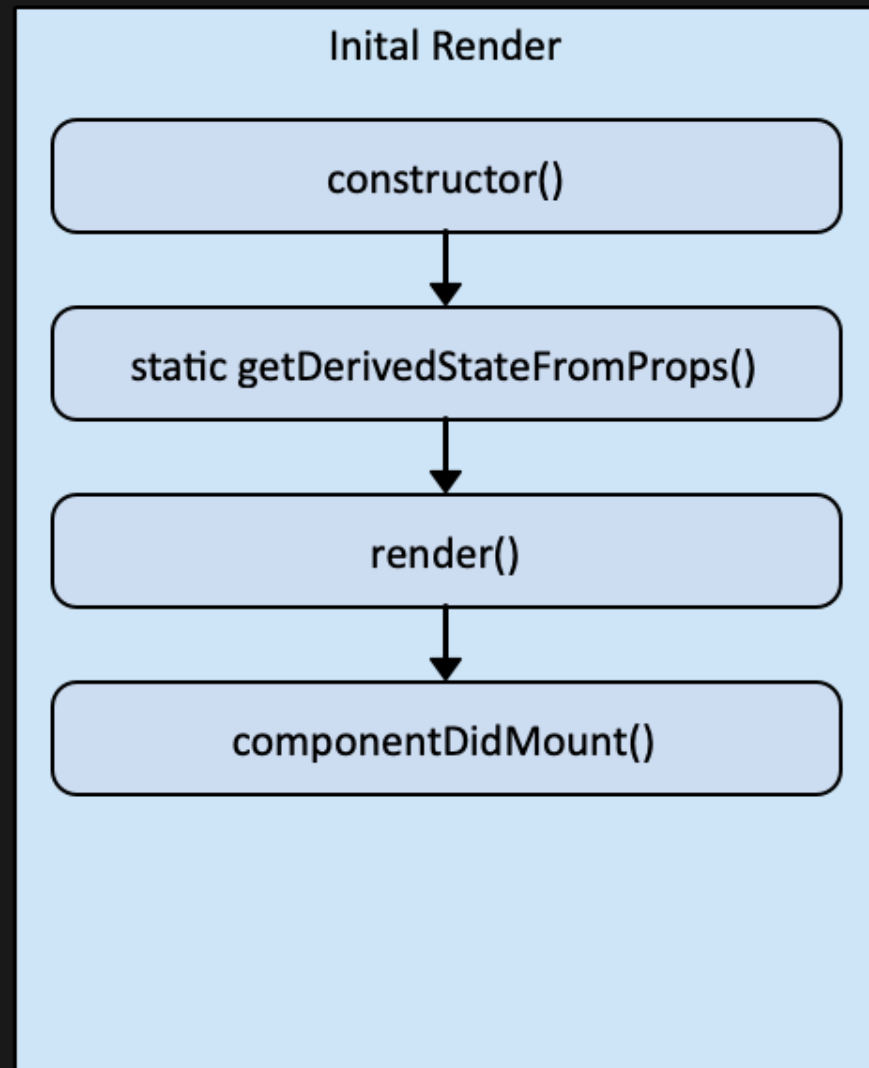
# COMPONENT LIFECYCLE

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

```
1 function users(fail) {
2   return new Promise((resolve, reject) => {
3     setTimeout(() => {
4       if (fail) {
5         reject('epic fail')
6       } else {
7         resolve({
8           users: [
9             { id: 0, name: 'First' },
10            { id: 1, name: 'Second' },
11            { id: 2, name: 'Third' },
12          ],
13        })
14      }
15    }, 2000)
16  })
17 }
```



# FETCHING DATA

```
1  class UserListContainer extends Component {
2    state = {
3      data: {
4        error: null,
5        loading: 'loading...',
6        users: [],
7      },
8    }
9
10   componentDidMount() {...}
11
12   render () {
13     return ( <UserList {...this.state.data} /> )
14   }
15 }
```

# LIFECYCLE METHOD

```
1  componentDidMount() {  
2    users().then(      // users(true) to reject Promise  
3      (result) => {  
4        this.setState({  
5          data: {  
6            error: null,  
7            loading: null,  
8            users: result.users,  
9          },  
10         })  
11       },  
12       (error) => {  
13         this.setState({  
14           data: {  
15             error: error,  
16             loading: null,  
17             users: this.state.users.
```

# UI COMPONENTS

```
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

```
1 import React, { useState, useEffect } from 'react'
2
3 function Example() {
4   const [count, setCount] = useState(0)
5
6   // Similar to componentDidMount and componentDidUpdate:
7   useEffect(() => {
8     document.title = `You clicked ${count} times`
9   })
10
11   return (
12     <div>
13       <p>You clicked {count} times</p>
14       <button onClick={() => setCount(count + 1)}>
15         Click me
16       </button>
17     </div>
18   )
19 }
```

# 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: lifecycle 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

```
1 // Passing a dependency array
2 useEffect(() => {
3   // ...
4 }, [a, b]) // Runs again if a or b are different
5
6 // Passing an empty dependency array
7 useEffect(() => {
8   // ...
9 }, []) // Does not run again
10
11 // Passing no dependency array at all
12 useEffect(() => {
13   // ...
14 }) // 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 be 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

```
1 function Toolbar (props) { // no need to pass down the theme
2   return (
3     <div>
4       <ThemedButton />
5     </div>
6   )
7 }
8
9 function ThemedButton () {
10   const theme = useContext(ThemeContext)
11
12   return (
13     <button style={{ background: theme.background, color: theme.foreground }}>
14       I am styled by theme context!
15     </button>
16   )
17 }
```

# 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

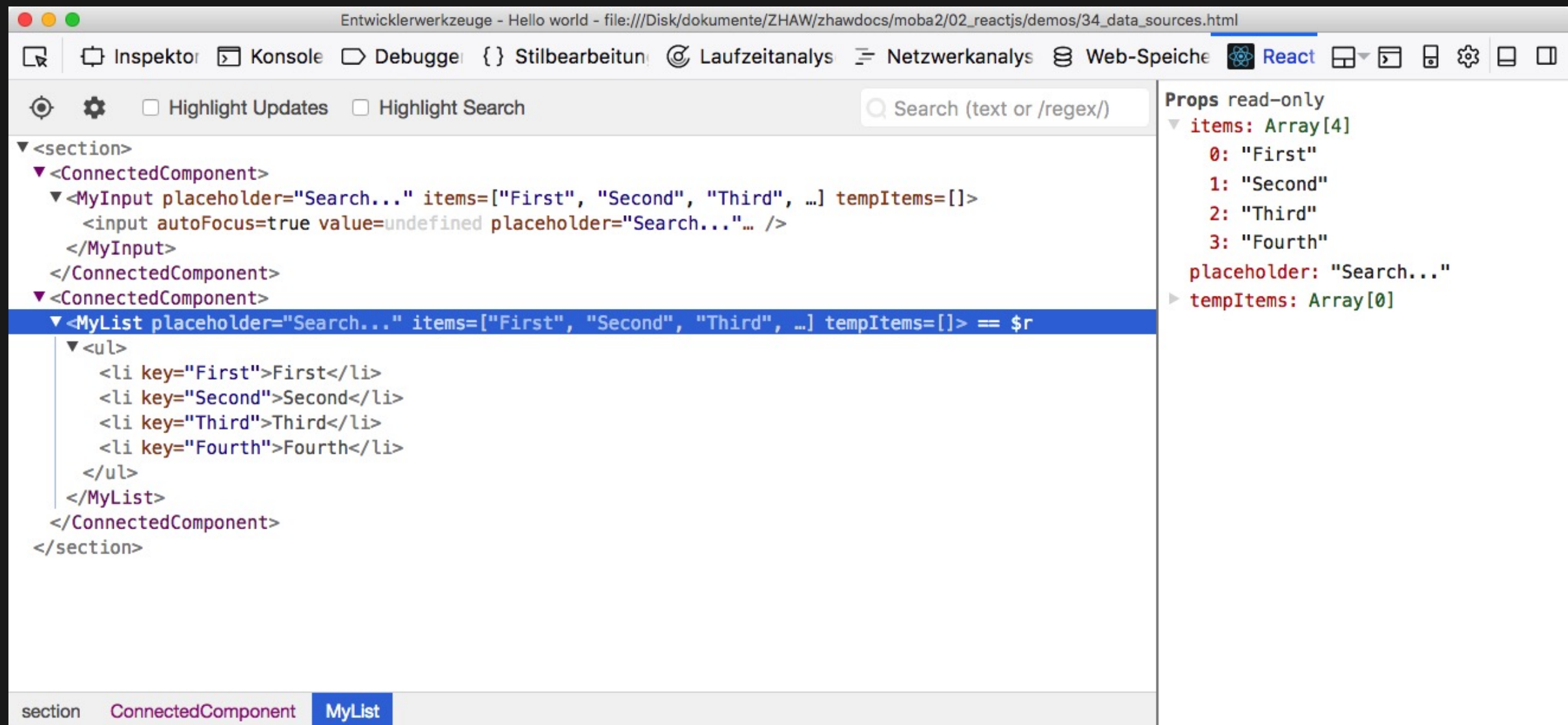
```
$ npx create-react-app hello-world  
$ cd hello-world/  
  
$ npm start  
Starting the development server...  
Compiled successfully!  
The app is running at: http://localhost:3000/  
  
Note that the development build is not optimized.  
To create a production build, use npm run build.
```

- 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



# 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](#)

