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

Speaker notes

With properties, it’s often a parent component that decides when to render the JSX

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> )
  }
}
```

Speaker notes

- The initial state of a class component should always be an object
- Don’t use an array or any other type as state
- In case of problems with class properties, alternatively set the state in the constructor
- Or use Babel to convert (`es7.classProperties`)

```
class MyComponent extends Component {

  constructor(props) {
    super(props)
    this.state = {
      first: false,
      second: true,
    }
  }

  render() { ... }
}
```

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

Speaker notes

- The component is first rendered with it’s default state
- After 3 seconds, `setState()` changes the state property values
- This change is reflected in the UI

The UI is declared once using JSX syntax. Over time, the state of the component is changed. All the DOM interactions are optimized and hidden from view.

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

Speaker notes

Therefore, the initial state of a component should always be an object

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')
)
```

Speaker notes

- In the example, we have an `appState` object
- Here we've moved state outside of the component

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>
    )
  }
}
```

Speaker notes

- Set as a class property called `defaultProps`
- Used if the component is rendered without JSX properties
- Static elements are relatively new to JavaScript
- In case of problems, add `defaultProps` like this:

```
MyButton.defaultProps = {
  disabled: false,
  text: 'My 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

Speaker notes

The convention is to use arrow function syntax to declare functional React components. However, it’s perfectly valid to declare them using traditional JavaScript function syntax, if that’s better suited to your style.

DEFAULTS IN FUNCTION COMPONENTS

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

Speaker notes

`defaultProps` can also be used with function components

```
// The function component doesn't care if the property
// values are the defaults, or if they're passed in from
// JSX. The result is the same.
const MyButton = ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
)

// The "MyButton" constant was created so that we could
// attach the "defaultProps" metadata here, before
// exporting it.
MyButton.defaultProps = {
  text: 'My Button',
  disabled: false,
}
```

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 }
```

Speaker notes

- Add some local state to a function component
- React will preserve this state between re-renders
- `useState` returns a pair:
    - the current state value
    - a function that lets you update it (doesn't merge)
- Argument of `useState`: the initial state
- The state doesn't have to be an object

Same example in a class component:

```
class Example extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      count: 0
    }
  }
  render () {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    )
  }
}
```

The function returned by `useState()` is similar to `this.setState()` in a class, except it doesn't merge the old and new state together.

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

Speaker notes

When an event is triggered and it hits the root DOM container, React maps the event to the handlers. If a match is found, it calls the handler. Finally, when the React component is removed, the handler is simply removed from the list of handlers.

None of these DOM operations actually touch the DOM. It’s all abstracted by a single event listener. This is good for performance and the overall architecture (keep the render target separate from application code).

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)

Speaker notes

Event Pooling (React < v17.0):

- For performance reasons, React allocated an *event instance pool*
- Reason: prevents garbage collection when a lot of events are triggered
- When a handler has finished running, the instance goes back into the pool and all of its properties are cleared (can cause problems when accessed asynchroneously)

OVERVIEW

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

COMPONENT LIFECYCLE

React components go through a lifecycle

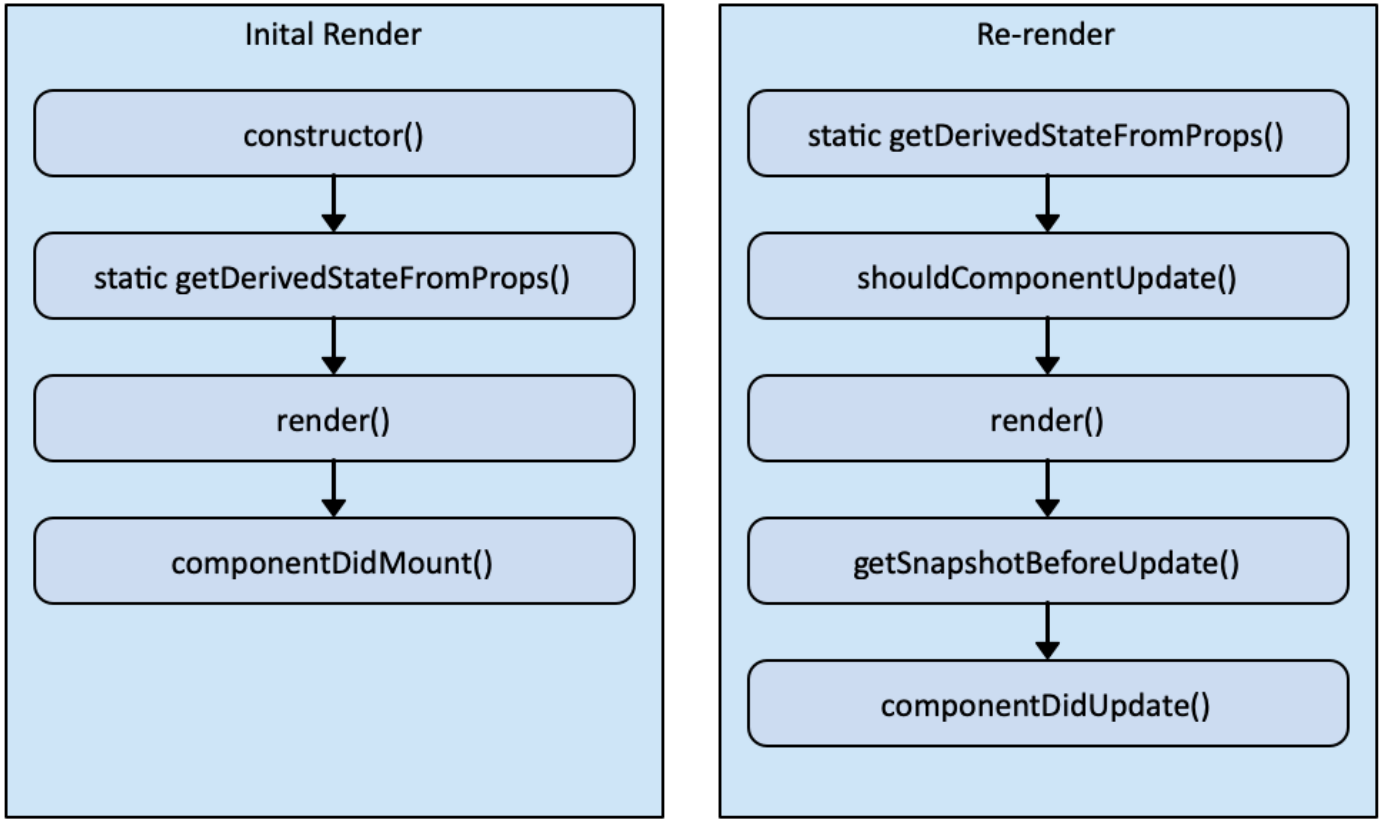
- Component is about to be mounted
- Component is rendered
- After the component has been mounted
- When the component is updated

... and so on

Speaker notes

Lifecycle events are yet another moving part, so you'll want to keep them to a minimum

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

```
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

```
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,
```

Speaker notes

```
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

Speaker notes

EXAMPLE

```
class MyList extends Component {

  state = {
    data: {
      items: new Array(5000)
        .fill(null)
        .map((v, i) => i),
    },
  }

  shouldComponentUpdate (nextProps, nextState) {
    return this.state.data.items !== nextState.data.items;
  }

  render () {
    const items = this.state.data.items
    return ( ... )
  }
}
```

For this to work, make sure the reference to items changes when the array is changed. Therefore, don’t change the array in place. This can be achieved for example with *Immutable.js*.

USING METADATA TO OPTIMIZE RENDERING

- Alternatively, metadata that's part of the API response is used
- The `modified` property determines whether the component should render

```
class MyUser extends Component {
  state = {
    modified: new Date(),
    first: 'First',
    last: 'Last',
  }

  shouldComponentUpdate (props, state) {
    return +state.modified > +this.state.modified;
  }

  render () { ... }
}
```

https://reactjs.org/docs/react-component.html#the-component-lifecycle

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 }
```

Speaker notes

The function passed to `useEffect` is going to be different on every render. This is intentional. In fact, this is what lets us read the count value from inside the effect without worrying about it getting stale. Every time we re-render, we schedule a different effect, replacing the previous one. In a way, this makes the effects behave more like a part of the render result — each effect “belongs” to a particular render.

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

Speaker notes

The idea is that you should be able to achieve some level of substitutability with this pattern. For example, a container could substitute its child component. Or, a child component could be used in a different container.

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} />
  )
}
```

Speaker notes

Same as a class component:

```
class MyContainer extends Component {

  // The container should always have an initial state
  state = { items: [] }

  // After the component has been rendered, fetch data
  componentDidMount() {
    fetchData()
      .then(items => this.setState({ items }))
  }

  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} />
  )
}
```

Speaker notes

The Toolbar component must take an extra "theme" prop and pass it to the ThemedButton. This can become painful if every single button in the app needs to know the theme because it would have to be passed through all components.

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>
  )
}
```

Speaker notes

Use a Provider to pass the current theme to the tree below. Any component can read it, no matter how deep it is. In this example, we’re passing "themes.light" as the current value.

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>
  )
}
```

Speaker notes

In a class component, assign a contextType to read the current theme context. React will find the closest theme Provider above and use its value. In this example, the current theme is "themes.light".

```
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render () {
    return (
      <button style={{
        background: this.context.background,
        color: this.context.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

Speaker notes

The current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree.

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

Speaker notes

`npx` is a npm package runner (x probably stands for eXecute). The typical use is to download and run a package temporarily or for trials.

`create-react-app` is an npm package that is expected to be run only once in a project's lifecycle. Hence, it is preferred to use npx to install and run it in a single step.

```
$npm run build

> hello-world@0.1.0 build /Users/burkert/Desktop/hello-world
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  47.16 KB  build/static/js/main.2b26bd27.js
  289 B     build/static/css/main.9a0fe4f1.css

The project was built assuming it is hosted at the server root.
To override this, specify the homepage in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage": "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may also serve it locally with a static server:
```

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

Speaker notes

For Atom, *language-babel* can be helpful
https://github.com/gandm/language-babel

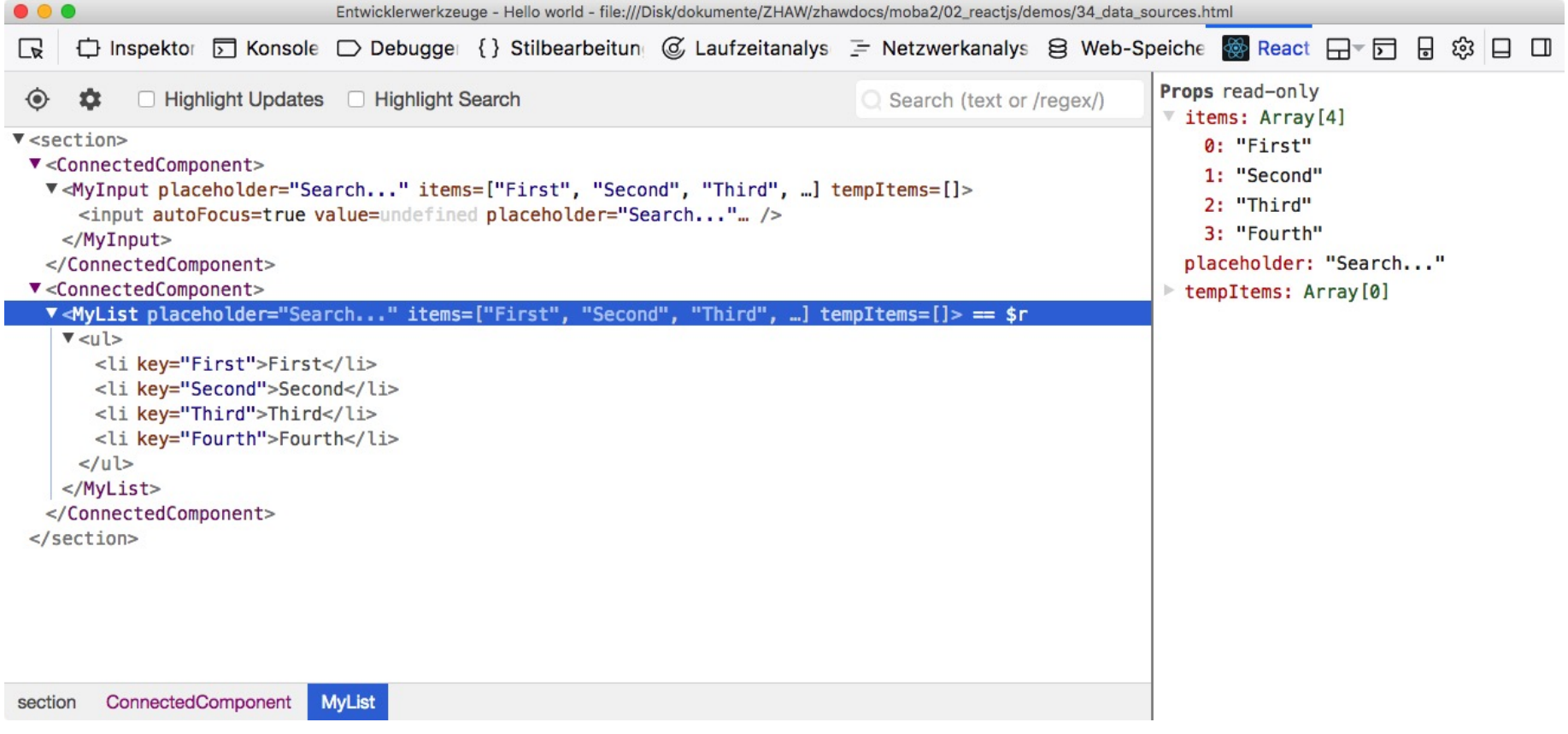
If you use Emmet in VSCode, JSX support can be added
https://dev.to/harender24/enable-jsx-support-in-vs-code-react-46od

From dev.to:

1. In VS Code, open any source folder and navigate to Settings: Ctrl +, on Windows
2. Go to the Workspace Settings tab.
3. Select Emmet from the Extensions selection on the left sidebar.
4. Select "Edit in settings.json"
5. In the newly opened settings.json file, add the following lines.

```
{
    "emmet.includeLanguages": {
       "javascript": "javascriptreact"
    }
}
```

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