

PV247

Martin Knapik

Content of Lecture

- React component types
 - What are different ways how to declare components and when to use which?
- React Lifecycle
 - What does life of React component look like, and how can we use it in our projects?
- Ref
 - How to integrate React with 3rd party libraries or invoke imperative calls on DOM elements?
- Immutability
 - Why is immutability important in React and how to include it seamlessly in our projects?
- Higher Order Components
 - How to reuse existing logic in project?
- Demo
 - Todo List

Functional Component - example

```
function HelloWorldComponent(props) {  
  return (  
    <div>  
      Hello {props.name}  
    </div>  
  );  
}
```

Functional Component


- Also called Stateless
- Function getting props on input and returning React element
- Pros:
 - Short declaration
 - Pure (no side effects) by design
- Cons:
 - Doesn't have state
 - Performance
 - Always rerenders even if props didn't change
 - Doesn't have access to lifecycle methods

Functional Component – arrow function

```
const HelloWorldComponent = (props) => (  
  <div>  
    Hello {props.name}  
  </div>  
);
```

Component Class - example

```
class HelloWorldComponent extends React.Component {  
  render() {  
    return (  
      <div>  
        Hello {this.props.name}  
      </div>  
    );  
  }  
}
```



Component Class

- Component is declared as Class extending *React.Component*
- React element is returned from *render()* method
- Methods need to be bound to context of class
- Pros:
 - Does have state
 - Has access to lifecycle methods
- Cons:
 - Performance
 - Always rerenders even if props didn't change
 - Declaration is longer than functional components

Pure Component Class - example

```
class HelloWorldComponent extends React.PureComponent {  
  render() {  
    return (  
      <div>  
        Hello {this.props.name}  
      </div>  
    );  
  }  
}
```


Pure Component Class

- Component is declared as Class extending *React.PureComponent*
- Same as Component Class
- Does shallow comparison on props/state change to evaluate whether to rerender
- Pros:
 - All the pros of Component Class
 - Performance
 - Rerenders only if props/state changes
 - Components have to be pure (no side effects)
- **Use whenever possible**

Ref

- Ref is reference to the DOM element / React component
- Created using method *createRef* and then assigning return value to element
- Escape hatch when we need imperative calls on Components and DOM Elements
 - Calls to Component methods
 - Usually can be rewritten declaratively
 - Integration with 3rd party libraries not written in React
 - Invoking calls on DOM elements (`input.focus()`, `audio.play()`, etc.)

Ref

```
class TextInput extends React.PureComponent {  
  constructor(props) {  
    super(props);  
    this.input = React.createRef();  
  }  
  
  render() {  
    return (  
      <input type="text" ref={this.input}/>  
    )  
  }  
}
```

React Component Lifecycle

- React Component has Lifecycle
 - It is created, and then inserted into DOM
 - Its props or state is changed and it is reevaluated and rerendered
 - It is removed from DOM

Lifecycle Methods

- React Components expose Lifecycle Methods
 - Each is called at certain specific phase of Components life
 - We can override these methods and inject our code into them to be executed at specified moment in life of method
 - We can do something that shouldn't be possible in declarative programming

Lifecycle Methods

- 10 total
- Most commonly used
 - render
 - constructor
 - componentDidMount
 - componentWillUnmount
 - shouldComponentUpdate

render

- Only required lifecycle method in Component Class
- Props + State => React Element

```
render() {  
  return (  
    <div>  
      Hello {this.props.name}  
    </div>  
  );  
}
```

constructor

- Used for initializing state and binding methods – otherwise not needed
- Used to create refs
- It needs to call *super(props)*

```
constructor(props) {  
  super(props);  
  state = {  
    index: 0,  
    show: false  
  };  
  this.toggleVisibility = this.toggleVisibility.bind(this);  
}
```


componentDidMount

- Called immediately when React Element is mounted into DOM
- Used for initialization
 - Initialization that requires DOM (e.g. focus() for inputs)
 - Data request (e.g. API call)
 - Subscriptions

componentWillUnmount

- Called before React element is removed from DOM
- Used for cleanup
 - Invalidating timers
 - Cancelling requests
 - Unsubscribing

shouldComponentUpdate

- Called before every but first render
- Used for performance optimization
 - PureComponent is overriding it automatically for shallow comparison
 - We can give custom logic here for better optimization
- Returns Boolean
 - If false then render is not called

Lifecycle Methods

- Overview + use cases
 - <https://reactjs.org/docs/react-component.html>
- As of React 16 some of lifecycle methods were made unsafe and replaced by alternative
 - In React 17 they will be removed

Immutability

- Every change to object has to return new object
- Mutability
 - Allows to change objects properties without changing its reference
 - Possible issues related to React
 - We can change object without changing reference to it
 - Might cause issue when detecting props/state change
 - Props can be changed by child components
 - Breaks 1-way data binding – component receives props, invokes callbacks

Immutability

- Spread operator/Object cloning
 - Performance issues with large structures

Immutability

- Immutable.js
 - Facebook library
 - Implements basic structure types
 - Array -> List
 - Object -> Map
 - Provides additional structures
 - Set
 - Record
 - Provides more methods to work with structure
 - (Almost) Every operation returns new object
 - Has mutable mode, shouldn't be used
 - Documentation + API reference
 - <http://facebook.github.io/immutable-js/docs/#/>

Higher Order Components

- Analogy to Higher Order Functions
- Higher Order Function
 - Function which takes function as argument or returns function
- Higher Order Component
 - Function which takes **component** as argument **and** returns **component**

Higher Order Components

```
const Text = (props) => ( // Component
  <div>{props.value}</div>
);

function giveValue(Component) { // Higher Order Component
  return class ValueProvider {
    render() {
      return (
        <Component {...this.props} value={42} />
      )
    }
  }
}

export const TextWithValue = giveValue(Text); // Wrapped component
```

Higher Order Components

- Useful for reusing component logic
 - Regular Components are reusing “markup”
- Wrap existing component for additional functionality
 - Drag and drop
 - Access to local storage
 - Autosave
- Many existing 3rd party community libraries are written as HoC
 - react-redux
 - react-dnd

Q&A