ReactJS Part 2

IMY 220 ● Lecture 18

Recap

Last time we looked at how we can create reusable groups of webpage elements using JSX. We created React components that take properties and nest each other to create scalable content and/or functionality

In this lecture we'll add interactivity to our components using component **state**

But first we'll look at adding and reacting to event handlers in HTML elements

Event handling

Handling events in JSX elements works slightly different than plain HTML

- React events are similar to HTML events, but use camelCase rather than lowercase, for example, onClick rather than onclick
- The event handler for a JSX element expects a function rather than a string

```
function sayYass(){
   alert("Yass");
}
class FunButton extends React.Component{
   render(){
      return (
         <button onClick={sayYass}> Click Me! </button>
      );
   }
}
```

Event handling

Default events have to cancelled explicitly. You cannot return false to prevent a default action, such as an anchor directing to a link or a form submitting, you must call preventDefault

```
function sayYass(e){
   alert("Yass");
   e.preventDefault();
}
class FunButton extends React.Component{
   render(){
      return (
         <a onClick={sayYass}> Click Me! </a>
      );
   }
}
```

Event handling

```
class FunButton extends React.Component{
   sayYass(e){
      alert("Yass " + this.props.name);
      e.preventDefault();
   }

   render(){
      return (
         <a onClick={this.sayYass}> Click Me! </a>
      );
   }
}
ReactDOM.render(
        <FunButton name="Diffie" />,
        document.getElementById("react-container")
);
```

Rather than using global functions, you generally want the functions being called by a component to be a member function of that class

Event handling

The example on the previous slide gives the following error:

TypeError: this is undefined

This is because sayYass() has not been **bound** to the class, which means it does not have access to the class's this

Event handling

One way to solve this is with a function called bind

"The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called."

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind

In other words: bind allows us to set what this refers to

```
this.x = 9;    // this refers to global "window" object here in the browser
var module = {
  x: 81,
  getX: function() { return this.x; }
};

module.getX(); // 81

var retrieveX = module.getX;
retrieveX();
// returns 9 - The function gets invoked at the global scope

// Create a new function with 'this' bound to module
// New programmers might confuse the
// global var x with module's property x
var boundGetX = retrieveX.bind(module);
boundGetX(); // 81
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind#Creating_a_bound_function

```
class FirstPerson{
        constructor(name, surname){
                this._name = name;
        }

        sayName(){
                console.log(`My name is ${this._name}`);
        }
}

class SecondPerson{
        constructor(name, surname){
                this._name = name;
        }

        sayName(){
                console.log(`My name is ${this._name}`);
        }
}
```

```
let p1 = new FirstPerson("Dipper");
let p2 = new SecondPerson("Mabel");

p1.sayName();
// Output: My name is Dipper

p1.sayName = p1.sayName.bind(p2);
p1.sayName();
// Output: My name is Mabel
```

Event handling

This solves our problem by allowing us to bind member functions to the class, which gives access to all the class's member functions and properties through the this keyword

To bind the function, we must first add a **constructor**. In React, the constructor of a component must always call the base constructor with its own props as a parameter, in other words:

```
constructor(props){
    super(props);
}
```

Event handling

Our constructor would thus look something like this

```
constructor(props){
    super(props);
    this.sayYass = this.sayYass.bind(this);
}
```

This allows us to access this.props.name inside the sayYass() function, since the function's this now refers to the class's this

```
class FunButton extends React.Component{
   constructor(props){
      super(props);
      this.sayYass = this.sayYass.bind(this);
   }
   sayYass(e){
      alert("Yass " + this.props.name);
      e.preventDefault();
   }

   render(){
      return (
         <a onClick={this.sayYass}> Click Me! </a>
      );
   }
}
ReactDOM.render(
        <FunButton name="Dipper" />,
        document.getElementById("react-container")
);
```

Component state

One of the advantages of implementing components as ES6 classes is that it allows us to make use of React's **state** functionality

State is similar to props, but has some features that are useful for creating components that provide interactivity and updateability

Similar to props, state is access through this.state. However, state has to be initialised as a JS object in the constructor

Component state

The following example simply initialises the component's state and renders a different output depending on the state

```
class ToggleButton extends React.Component{
   constructor(props){
      super(props);
      this.state = {toggled: true};
   }

   render(){
      return (
         <button> {this.state.toggled ? "On" : "Off"} </button>
      );
   }
}     // Output: <button> On </button>
```

Component state

State is updated with the setState() function, which accepts a new JS object containing the key-value pair(s) of the state(s) you want to update

```
// (Using the existing ToggleButton class)
toggle(){        // toggled is switched between true and false
   this.setState({toggled: !this.state.toggled});
}
```

We can call this function in our button's onClick handler (if we remember to bind it to our class)

```
// Inside render()
return (
   <button onClick={this.toggle}>
      {this.state.toggled ? "On" : "Off"}
   </button>
);
```

```
class ToggleButton extends React.Component{
    constructor(props){
        super(props);
        this.toggle = this.toggle.bind(this);
        this.state = {toggled: true};
    }

    toggle(){
        this.setState({toggled: !this.state.toggled});
    }

    render(){
        return (
            <button onClick={this.toggle}>
              {this.state.toggled ? "On" : "Off"}
            </button>
        );
    }
}
```

Component state

The example on the previous slide creates a button that toggles its text between “On” and “Off” when you click on it

You’ll notice that we didn’t have to call render() again when updating the state for the new state to reflect in the element. That is part of how React handles updates to the DOM.

“You tell React what state you want the UI to be in, and it makes sure the DOM matches that state.”

https://reactjs.org/docs/faq-internals.html

Component state

In other words, when you modify the component state, it automatically updates the UI as efficiently as possible

This only works when you call setState. Modifying the state variables directly (for example: this.state.toggled = true) does not update the changes to the UI and defeats the purpose

Read up more on asynchronous updates, merging state variables, etc. here: https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

Refs

Sometimes we want a component to be able to interact with its child elements, for example, to collect values from inputs, set focus to a particular input, etc.

React's built-in method of doing this is through **refs**. Refs give us a way to access child components.

Refs are created with the React.createRef() function*

*As of v16.3

Refs

As a simple example, we're going to create a form that alerts the values of its two inputs

To do this, we'll need to create two refs: one for each input. We do this in the constructor

```
class AddPersonForm extends React.Component{
      constructor(props){
            super(props);
            this.nameInput = React.createRef();
            this.surnameInput = React.createRef();
      }
```

Refs

Now we can attach those refs to React elements in the render() function.

(Since we want something to happen when we submit our form, we're also adding an event handler to the form's onSubmit)

```
render(){
   return (
      <form onSubmit={this.submit}>
         <input type="text" ref={this.nameInput} /> <br/>
         <input type="text" ref={this.surnameInput} /> <br/>
         <input type="submit" value="Send" />
      </form>
   );
}
```

Refs

Now that we have attached refs to our elements, we can refer to them with this.(refName).current

Since our refs are input elements, we can use this.(refname).current.value to get their values

```
let name = this.nameInput.current.value;
let surname = this.surnameInput.current.value;
// After submitting our forms, these should contain the values
// entered into the input boxes
```

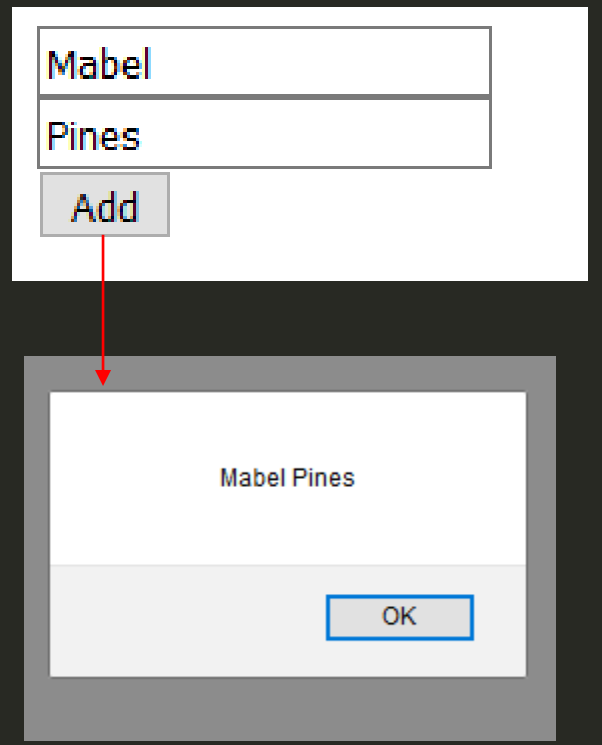
Refs

The only things we still have to do are to create a function that gets the values of the input boxes using the refs

…and to bind it so we can assign the form's onSubmit event handler to our function…

…and to prevent the form from submitting

```
class AddPersonForm extends React.Component{
    constructor(props){
        super(props);
        this.submit = this.submit.bind(this);
        this.nameInput = React.createRef();
        this.surnameInput = React.createRef();
    }

    submit(e){
        e.preventDefault();
        let name = this.nameInput.current.value;
        let surname = this.surnameInput.current.value;
        alert(`${name} ${surname}`);
    }

    render(){
        return (
            <form onSubmit={this.submit}>
                <input type="text" ref={this.nameInput} /> <br/>
                <input type="text" ref={this.surnameInput} /> <br/>
                <input type="submit" value="Add" />
            </form>
        );
    }
}
```



Refs (older versions)

Note that older versions of React used a different method of using refs, called string refs, which simply used a string identifier for each element and used this.refs.(stringIdentifier) to access the element

```
// Inside render()
<input type="text" ref="name" />


// Inside submit()
let name = this.refs.name.value;
```

This method is being deprecated in favour of createRef()

PersonList example

Let's extend our <PersonList /> component to use our <AddPersonForm /> for adding new people to the list

We want the <PersonList /> to contain a list of <Person /> components as well as an <AddPersonForm /> component for adding new people to the list

Therefore, we will keep track of the list of people in the state of the PersonList class and add functionality for updating this state

PersonList example

```
class PersonList extends React.Component {
   constructor(props){
      super(props);
      // If PersonList is created with a people={arrayOfPeople}
      // use that array, otherwise initialise a blank array
      this.state = {people: this.props.people || []};
      this.addPerson = this.addPerson.bind(this);
   }

   addPerson(name, surname){
      let people = this.state.people;
      people.push({name, surname});
      this.setState(people);
   }
```

PersonList example (quick aside)

The addPerson() function can actually be shortened using the spread operator and object literal enhancement…

```
addPerson(name, surname){
        this.setState({people: [...this.state.people, {name, surname}]});
}
```

…since the old array is included via the spread operator and object literal enhancement shortens {name: name, surname: surname} to

{name: surname}

PersonList example

We want the <PersonList /> to display an <AddPersonForm /> which calls PersonList's addPerson() function when it is submitted

To do this, we pass the addPerson() function as a prop inside the render() function of the PersonList class. (Note that this is not an event handler, we're simply passing a function as a prop)

```
// Inside PersonList's render()
<div>
   <AddPersonForm onNewPerson={this.addPerson} />
</div>
```

PersonList example

Now we have to call the function passed as a prop inside the AddPersonForm class

Inside our existing submit() function (from the previous example) we simply call the function passed as a prop with the name and surname refs as parameters

```
submit(e){
    e.preventDefault();
    let name = this.nameInput.current.value;
    let surname = this.surnameInput.current.value;
    this.props.onNewPerson(name, surname);
}
```

PersonList example

Submitting the form element of AddPersonForm now calls its onNewPerson() prop, which is also PersonList's addPerson() function…

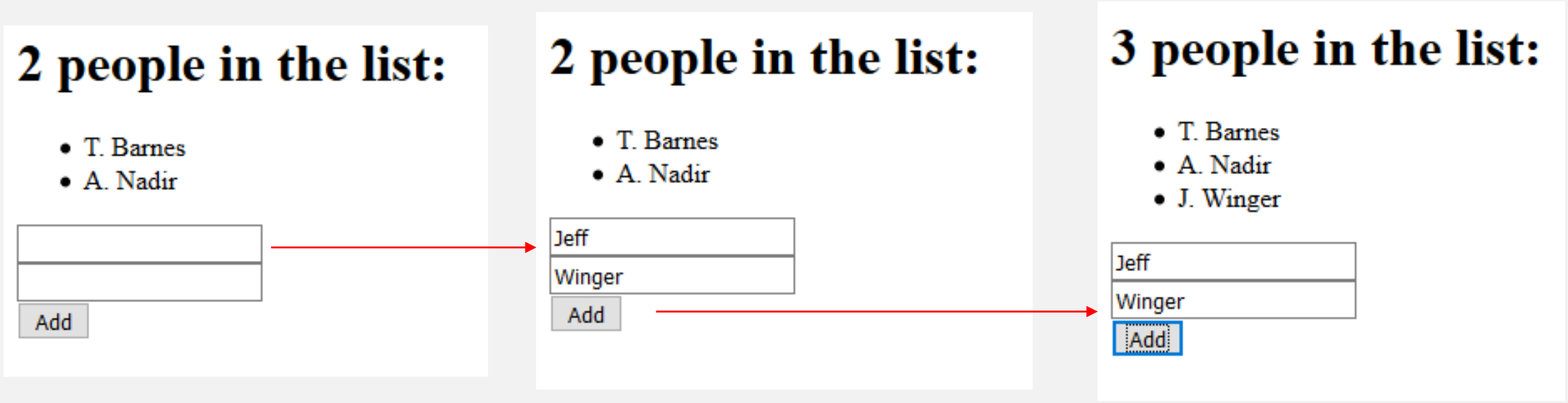
…which updates the state of PersonList using setState…

…which automatically updates the UI based on PersonList's state

```
class Person extends React.Component {
    render(){
        return (
            <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
        );
    }
}

class AddPersonForm extends React.Component{
    constructor(props){
        super(props);
        this.submit = this.submit.bind(this);
        this.nameInput = React.createRef();
        this.surnameInput = React.createRef();
    }

    submit(e){
        e.preventDefault();
        let name = this.nameInput.current.value;
        let surname = this.surnameInput.current.value;
        this.props.onNewPerson(name, surname);
    }

    render(){
        return (
            <form onSubmit={this.submit}>
                <input type="text" ref={this.nameInput} /> <br/>
                <input type="text" ref={this.surnameInput} /> <br/>
                <input type="submit" value="Add" />
            </form>
        );
    }
}
```

```
class PersonList extends React.Component {
    constructor(props){
        super(props);
        this.state = {people: this.props.people};
        this.addPerson = this.addPerson.bind(this);
    }

    addPerson(name, surname){
        this.setState({people: [...this.state.people, {name, surname}]});
    }

    render(){
        return (
            <div className="container">
                <h1>
                    {this.state.people.length ? this.state.people.length : "No"}
                    {this.state.people.length == 1 ? "person" : "people"} in the list:
                </h1>
                <ul>
                    {this.state.people.map( (person, i) => <Person key={i} person={person} /> )}
                </ul>
                <div>
                    <AddPersonForm onNewPerson={this.addPerson} />
                </div>
            </div>
        );
    }
}

var peopleList1 = [
    {name: "Troy", surname: "Barnes"},
    {name: "Abed", surname: "Nadir"}
];

ReactDOM.render(
    <PersonList people={peopleList1} />,
    document.getElementById('root')
);
```

PersonList example - result



Default props

If no prop value is given to a component expecting a prop, it will throw an error.

For example, if we create a <Person /> without a person prop…

```
{this.state.people.map( (person, i) => <Person key={i} /> )}
```

…it will give this.props.person is undefined

Default props

We can prevent errors like this by establishing default prop values, in other words, values that are used for props that have not been set

There are two ways of setting default props:

- outside the class definition

- inside the class definition with a static function

Default props

Setting defaultProps outside class definition

```
class Person extends React.Component{
   // class definition goes here
}

Person.defaultProps = {
   person: {name: "Default", surname: "Person"}
}
```

Default props

Setting defaultProps inside class definition with a static function

```
class Person extends React.Component{
   static defaultProps = {
      person: {name: "Default", surname: "Person"}
   }

   // rest of class definition goes here
}
```

Default props

Using defaultProps and additional code to reflect state, we can make our PersonList class more robust

First, let's initialise the people props to an empty array by default

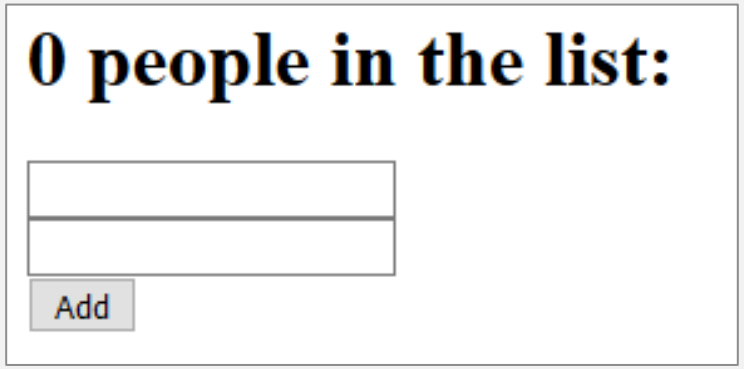
```
class PersonList extends React.Component{
   static defaultProps = {
      people: []
   }

   // rest of class definition goes here
}
```

Default props

Now if we render it without giving it a list of people…

```
ReactDOM.render(<PersonList />, document.getElementById('root'));
```

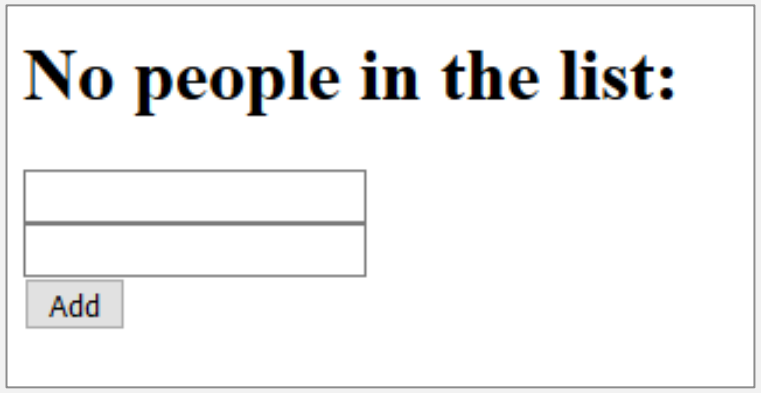
…instead of giving an error, it will show the following

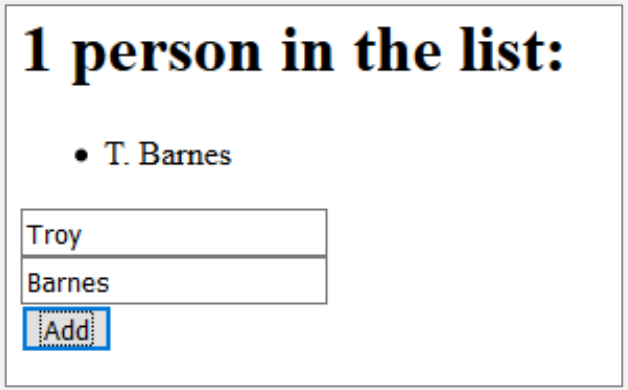


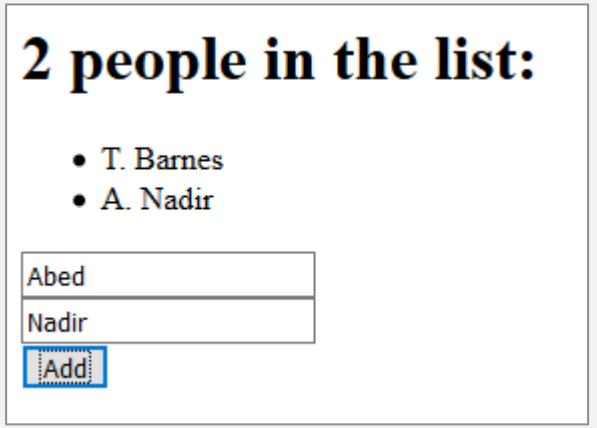
Default props

We can make it more descriptive by changing how the heading gets displayed based on the state of the PersonList

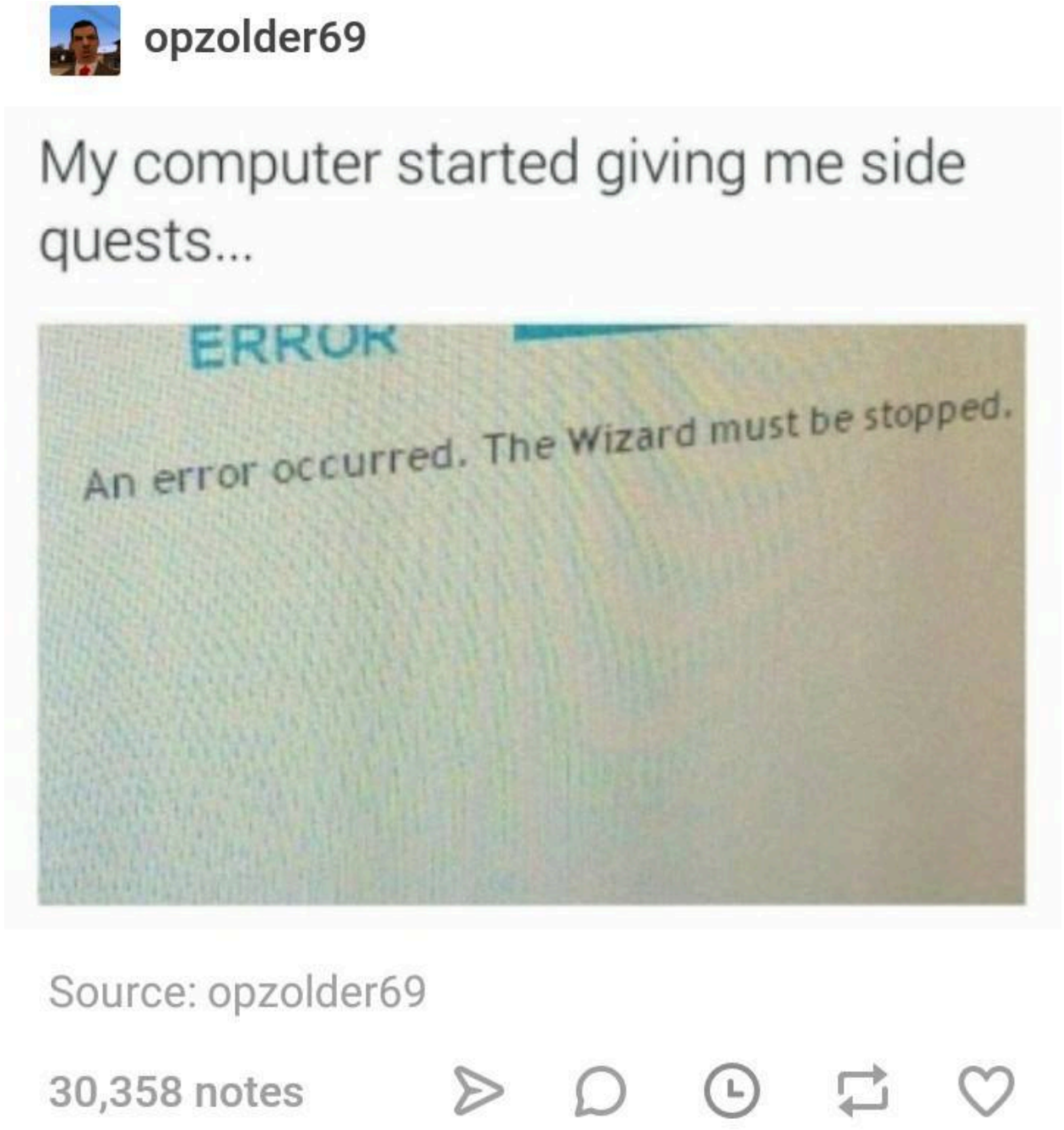
```
<h1>
   {this.state.people.length ? this.state.people.length : "No"}
   {this.state.people.length == 1 ? "person" : "people"} in the list:
</h1>
```







The End


opzolder69
My computer started giving me side quests...
ERROR
An error occurred. The Wizard must be stopped.
Source: opzolder69
30,358 notes

References

Banks, A. & Porcello, E. 2017. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, Inc.

https://reactjs.org/docs/

https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/