ReactJS workshop

Devpoint Labs - Dave Jungst

Assumptions

- You know HTML
- You know some JavaScript
- You understand including JS/CSS libraries in an HTML file
- You are familiar with the DOM
- You know how to use an IDE
- This is not an HTML / JS beginner workshop.

Covered VS Not covered

- Including ReactJS in a project
- Creating ReactJS components
- state / props
- Child components
- parent / child communication
- ES2015

- HTML
- JS Basics
- In depth ES2015
- Databases
- Web requests (AJAX)
- Separation of concerns
- Servers

What is ReactJS?

ReactJS is a JavaScript **library** (not **framework**) that was created by Facebook for rendering data as HTML elements.

ReactJS is just the UI. ReactJS is often used as the **V** in M**V**C as it makes no assumptions about the rest of your stack.

ReactJS Features

- 1-way data flow
  - Properties are set as immutable data objects. Data flows down and state keeps track of the current state of the component at any given time.
- Virtual DOM
  - The DOM is preserved when writing ReactJS. React keeps track of the DOM state in memory and diffs the DOM to determine which elements to update in the component.
- JSX
  - A JavaScript extension that allows you to write an XML/HTML like syntax in JS

Why ReactJS

- Speed!
  - ReactJS is fast and delivers a better user experience
  - No detectable client side latency
- Structured JS
  - No more spaghetti code in JS
- Modern web experience
- Happy developers
  - Writing JS is fun again

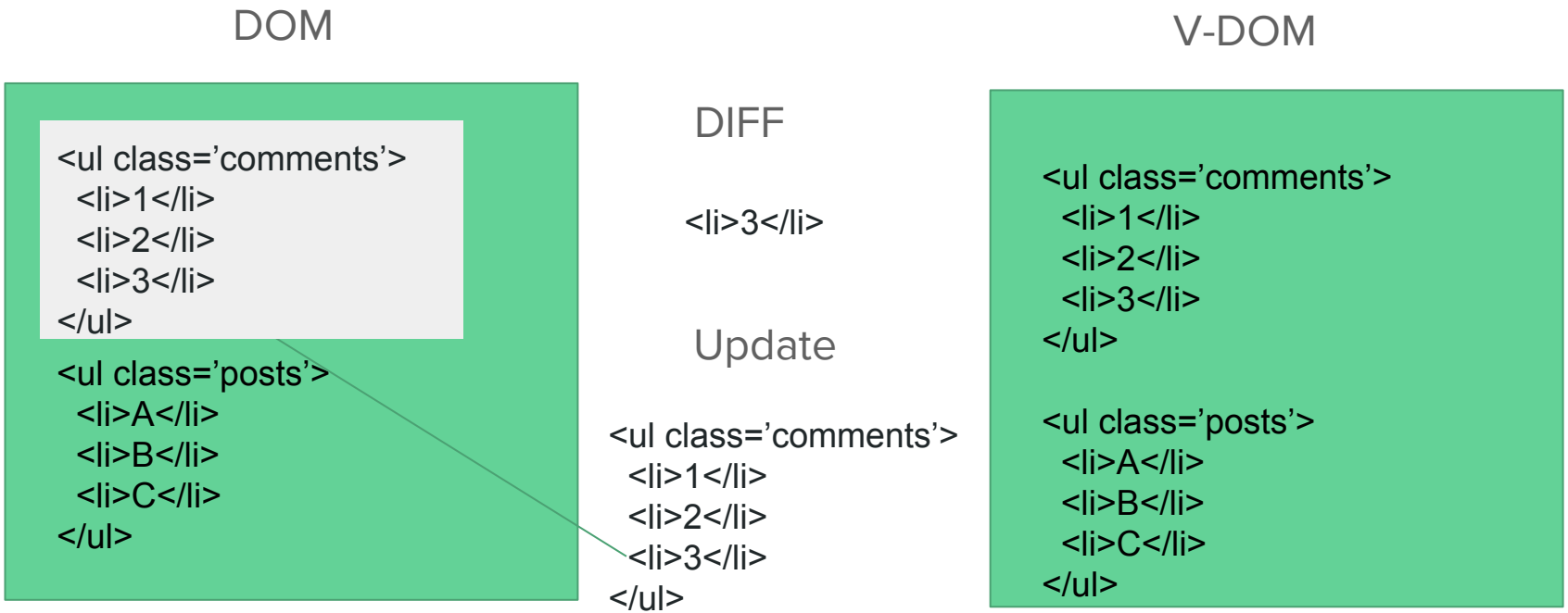
Other frameworks e.g. Angular, Ember, etc...

Angular, Ember, and any other “flavor of the week” JavaScript frameworks are MVC frameworks. ReactJS is just for the User Interface (Just the View layer). ReactJS is a library not a framework.

React is a very small library.

You can create React components in an existing app very quickly and easily without a full rewrite of the app.

Virtual DOM


DOM
<ul class='comments'>
1
2
3

<ul class='posts'>
A
B
C

DIFF
3
Update
<ul class='comments'>
1
2
3

V-DOM
<ul class='comments'>
1
2
3

<ul class='posts'>
A
B
C

Props & State

Props are short for Properties. **Props** are passed down to components. **Props** are immutable and can be thought of as the initial **state** of the component.

State is the current **state** of the component at any given point in time. Whenever the **state** is changed **render** is called and the virtual DOM diffs the DOM to update the component.

Because we want the V-DOM to do the heavy lifting and we want the DOM preserved it is bad practice to manipulate the DOM with libraries like JQuery.

Including ReactJS

For the sake of this tutorial we will build our entire app in a single file called index.html. In practice we would separate our HTML / CSS / JS.

index.html

There are many ways to include / install ReactJS. In this workshop I will just use the CDN to include the JS files needed in the <head>. I will also be including materializecss, a css framework just to make things look a little nicer but this is in no way required for ReactJS.

I am also including Babel so that we can write ES2015 (ES2015 and React play very nicely together)

index.html

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.5/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.5/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/marked/0.3.2/marked.min.js"></script>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.5/css/materialize.min.css">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.5/js/materialize.min.js"></script>
  </head>
  <body>
    <div id="content" class='container'></div>
    <script type="text/babel">
      //Our react code will go here
    </script>
  </body>
</html>
```

First React Component

```
<script type="text/babel">
  class List extends React.Component {
    constructor(props){
      super(props);
    }
    render() {
      return(<div className='center container'>
                <h1>Hello World</h1>
              </div>);
    }
  }

  ReactDOM.render(<List />, document.getElementById('content'));
</script>
```

New terminology and functions

class List extends React.Component

- Using ES2105 we are creating a class called List and extending React. Component

constructor(props)

- In our class we set up a constructor, this is calle when an instance of the class is instantiated. We allow our controller to take in properties.

super(props)

- Since we are extending React we just pass the props up to the React. Component constructor.

render

- React components return and HTML component through the render function. Each React component can have only 1 render function. Render is called when the component is mounted and any time the components state changes. All react components should be wrapped in a single div.

ReactDOM.render(<List />, document.getElementById('content'))

- This is not part of our component this is how we tell our app wich component to render and what element to attach it to.

className

- JSX is not HTML. We can't use keywords like class or for so HTML classes are replaced with className

State

I am going to rewrite the component to actually be meaningful instead of just writing Hello World to the browser.

```
class List extends React.Component{
  constructor(props){
    super(props);
    this.state = { items: [], complete: 0, nextID: 0 };
  }
  render(){
    return(<div className='center'>
            <h1> To Do </h1>
            <form onSubmit={this.addItem}>
              <input placeholder='add item' type='text' ref='name' />
            </form>
            <hr />
            <div className='row'>
              {this.items()}
            </div>
           </div>)
  }
}
```

this.state = { items: [], complete: 0, nextID: 0 }

- Calling this in the constructor sets the default state of the component when it is loaded. We are creating an object with 3 keys. Items will be an array to hold all of the items on the to do list. Complete will keep track of how many items we have completed. nextID will be used to add ID's dynamically to items.

<form onSubmit={this.addItem}

- When the form is submitted we are calling a function called addItem (this doesn't exist yet we need to write it. The { } are called expressions. This is used for writing logic inside of our JSX objects. e.g. className={ 1 + 2 } will result in an element with a class of 3

<input type='text' placeholder='add item' ref='name' />

- The only new thing here is ref. This is a way to keep track of dom elements.

```
<div className='row'>

 { this.items() }

</div>
```

Inside of this div we call a method called items (this does not exist yet we still need to write the function). Since we are using the ( ) syntax this function will be called during the execution of render.

Now to write out the methods that we stubbed in items and addItem

```
constructor(props){
  super(props);
  this.state = { items: [], complete: 0, nextID: 0 };
  this.items = this.items.bind(this);
  this.addItem = this.addItem.bind(this);
}
items(){
  let items = [];
  this.state.items.map( item => {
    items.push(<p>{item.name}</p>);
  });
  return items;
}
nextID(id){
  return id += 1;
}
addItem(e){
  e.preventDefault();
  let items = this.state.items;
  items.unshift({ name: this.refs.name.value, complete: false, id: this.state.nextID });
  this.refs.name.value = null;
  this.setState({ items: items, nextID: this.nextID(this.state.nextID) });
}
```

functions that will use the keyword **this** need to be bound to this in the constructor

this.items = this.items.bind(this)

The items function loops over all of the items in **this.state** and returns a **<p>** displaying the items name.

nextID was written as a helper function to return the **nextID + 1** so that all of our items have unique id’s

In the addItems function we pass in **(e)** but below onSubmit we are not passing in **e**.

On a listener the first element is the event that triggered it. We don't really want to submit a form so **e.preventDefault()** is called.

items.unshift({name: this.refs.name.value, complete: false, id: this.state.nextID})

- using unshift will push our item on to the front of the array. this.refs.name is the input since we set it to ref name.

this.refs.name = null

- This will clear out the input box

Finally we set the **state** of the app to be all of the items + the new item, and we increment **nextID**

Child components

This code works fine but it would be nice if we actually had a child component instead of just rendering items as **<p>** tags. Child components have their own **state**, their own **props** and their own **render**. Since the goal is to create composable components we will turn Items into a child component of list.

First we will create another class that extends **React.Component** called Item

```
class Item extends React.Component{
  constructor(props){
    super(props);
  }
  render(){
    let id = `complete-${this.props.id}`;
    return(<div className='col s12'>
             <div className='col m8'>
               {this.props.name}
             </div>
             <div className='col m3'>
               <input type="checkbox" id={id} defaultChecked={this.props.checked} />
               <label htmlFor={id}>Complete?</label>
             </div>
             <div className='col m1'>
               X
             </div>
           </div>);
  }
}
```

We can use string interpolation in ES2015 by using backticks and wrapping variables or expressions in **${ }**

let world = ‘Earth’

ES2015: `Hello ${world}`

ES5: ‘Hello ‘ + world

{this.props.name}

- our child component is stateless so we can just print the props.name instead of state.name. The state of the props passed in shouldn’t change on this component. This will allow the parent object to update the component by passing in different properties.

We use **defaultChecked=** instead of **checked=**. **defaultChecked** and **defaultValue** will create elements with Read / Write access. checked and value will create READONLY inputs!!!!

We also can’t use the keyword “for” so on the label we use **htmlFor**

Update the items function

Now that we have an Item component we can actually create items on the fly.

```
items(){
  let items = [];
  this.state.items.map( item => {
    items.push(<Item key={`item-${item.id}`} name={item.name} complete={item.complete} id={item.id}/>);
  });
  return items;
}
```

Instead of creating **<p>** tags we just create Items we pass in name complete and id as properties.

We also pass a key. When looping to create components React needs a way to keep track of them. A key must be unique!!!!!

```
class Item extends React.Component{
  constructor(props){
    super(props);
    this.state = { checked: false, style: {} };
    this.toggleChecked = this.toggleChecked.bind(this);
  }
  toggleChecked(){
    let complete = !this.state.checked;
    let style = {};
    if (complete) {
      style = {
        textDecoration: 'line-through',
        color: 'grey'
      }
    }
    this.setState({ checked: complete, style: style });
  }
  render(){
    let id = `complete-${this.props.id}`;
    let color = this.state.checked ? 'green lighten-5' : 'grey lighten-5';
    return(
      <div className={`${color} col s12`}>
        <div className='col m8' style={this.state.style}>
          {this.props.name}
        </div>
        <div className='col m3'>
          <input onChange={this.toggleChecked} type='checkbox' id={id} defaultChecked={this.props.checked}
          />
          <label htmlFor={id}>Complete</label>
        </div>
        <div className='col m1'>
          X
        </div>
      </div>
    )
  }
}
```

The background color will change depending on the state

Insert inline style

Call a method when the checkbox is checked / unchecked

There is a lot going on in the previous slide. Here is the high level.

We set a default state of checked for an item.

When a checkbox is clicked we change the state to the opposite of whatever it currently is.

If an item is checked we add some inline style to line through the text and change the text color.

We also set the background color of the element depending on the state of checked..

Parent child communication

When an item's checked state is updated we should tell the parent component about it so that the dataset can be updated to reflect the current state. If we wanted to have a complete count complete / number of items it would not update when an item was checked unless the dataset was updated to reflect this.

We have communicated from Parent to child by using **props**.

To communicate from child to parent we can use **callback** functions passed as **props**

Callbacks

callbacks are a way for a child to call a method on the parent allowing communication from child to parent.

When an item is updated or deleted we can use **callbacks** to tell the parent what happened.

The parent can then update the items array to reflect the current **state** which will force the parent to re-render using the most up to date dataset

Add some methods to the List constructor

```
class List extends React.Component{
  constructor(props){
    super(props);
    this.state = { items: [], complete: 0, nextID: 0 };
    this.items = this.items.bind(this);
    this.addItem = this.addItem.bind(this);
    this.completeCount = this.completeCount.bind(this);
    this.refreshCount = this.refreshCount.bind(this);
    this.deleteItem = this.deleteItem.bind(this);
  }
```

Create the new functions

```
completeCount(){
  let complete = this.state.complete;
  return(<p>{`${complete}/${this.state.items.length} complete`}</p>)
}
refreshCount(id, checked){
  let itemArray = this.state.items;
  let index = itemArray.findIndex( x => x.id === id);
  let item = itemArray.splice(index, 1)[0];
  item.complete = checked;
  let complete = this.state.complete;
  if (checked) {
    itemArray.push(item);
    complete++;
  } else {
    itemArray.unshift(item);
    complete--;
  }
  this.setState({complete: complete, items: itemArray })
}
deleteItem(name){
  let itemArray = this.state.items;
  let index = itemArray.findIndex( x => x.id === name);
  let checked = itemArray[index].complete;
  let complete = checked ? (this.state.complete - 1) : this.state.complete;
  itemArray.splice(index, 1);
  this.setState({ items: itemArray, complete: complete, nextID: this.nextID(this.state.nextID) })
}
```

completeCount()

- Returns a <p> tag of # of completed items / # of items

refreshCount()

- updates the count of completed items. If an item is checked it moves the items to the end of the array. If it is unchecked it moves it to the front of the array.

deleteItem()

- Updates the complete count if it needs to and then removes the item from the items array

update the List render to show the deleteCount

```
render(){
  return(<div className='center'>
          <h1> To Do </h1>
          <form onSubmit={this.addItem}>
            <input placeholder='add item' type='text' ref='name' />
          </form>
          {this.completeCount()}
          <hr />
          <div className='row'>
            {this.items()}
          </div>
        </div>)
}
```

Finally pass the callbacks as props to Item

```
items(){
  let items = [];
  this.state.items.map( item => {
    items.push(<Item key={`item-${item.id}`} deleteItem={this.deleteItem} refreshCount={this.refreshCount} name={item.name} complete={item.complete} id={item.id}/>);
  });
  return items;
}
```

Callbacks are passed as properties just like name, complete, item, and key

Now the child can call methods on the parent by calling

this.props.someMethodName()

Item component

First update **toggleChecked** to call the callback

```
toggleChecked(){
  let complete = !this.state.checked;
  this.setState({ checked: complete });
  this.props.refreshCount(this.props.id, complete);
}
```

Call the delete inline with a lambda

We can call **this.props.deleteItem(this.props.id)** without calling another function inside of the Item.

If we put it in an onClick and passed an ID the method would fire while the HTML was loading because the JSX parser sees the **( )** and tries to execute the method.

We can prevent this by creating an anonymous function with syntax like...

( ) => someMehtod(param)

```
<div onClick={() => this.props.deleteItem(this.props.id)} className='col m1'>
  X
</div>
```

Challenge Yourself!

This is just the beginning. There is so much more to learn about ReactJS but the concepts taught in this workshop are very important to master before moving on.

Create a Sticky-Note App

You should be able to:

Add notes

Delete Notes

BONUS:

Edit Notes

Move notes around by priority