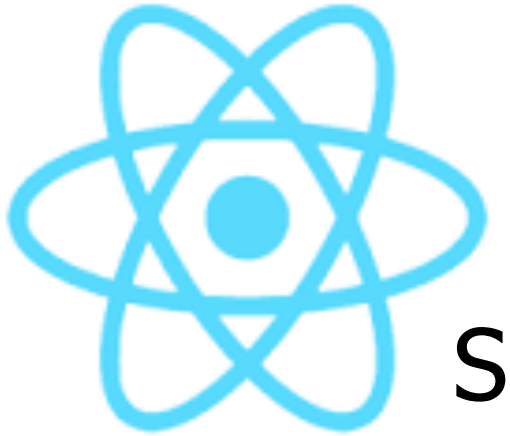


React

Say "No" to Complexity!

Mark Volkmann, Object Computing, Inc.
Email: mark@ociweb.com
Twitter: @mark_volkmann
GitHub: mvolkmann
Website: http://ociweb.com/mark

https://github.com/mvolkmann/react-examples

OCI | WE ARE SOFTWARE ENGINEERS.

What is OCI?

- New home of **Grails**,
  "An Open Source high-productivity framework
  for building fast and scalable web applications"
- Open Source Transformation Services, IIoT, DevOps
- offsite development, consulting, training
- **handouts** available (includes Grails sticker)

Overview ...

- Web app library from Facebook
  - http://facebook.github.io/react/
- Focuses on view portion
  - not full stack like other frameworks such as AngularJS and EmberJS
  - use other libraries for non-view functionality
    - some are listed later
- "One-way reactive data flow"
  - UI reacts to "state" changes
  - not two-way data binding like in AngularJS 1
    - what triggered a digest cycle?
    - should I manually trigger it?
  - easier to follow flow of data
    from events
    to state changes
    to component rendering

As of 1/2/16, **React** was reportedly **used by** Airbnb, Atlasssian, Capitol One, Codecademy, Coursera, Dropbox, Expedia, **Facebook**, **Firefox**, Flipboard, HipChat, IMDb, **Instagram**, Intuit, Khan Academy, NHL, **Netflix**, Paypal, Reddit, Salesforce, Squarespace, Tesla Motors, New York Times, Twitter, Uber, **WhatsApp**, Wired, Wordpress, Yahoo, Zendesk, and many more.
Source: https://github.com/facebook/react/wiki/Sites-Using-React

Facebook uses React more than Google uses Angular.

… Overview

- Can use in existing web apps that use other frameworks
  - start at leaf nodes of UI and gradually work up,
    replacing existing UI with React components
- Defines components that are composable
  - whole app can be one component that is built on others
- Components get data to render from "state" and/or "props"
- Can render in browser, on server, or both
  - ex. could only render on server for first page
    and all pages if user has disabled JavaScript in their browser
  - great article on this at https://24ways.org/2015/universal-react/
- Can render output other than DOM
  - ex. HTML5 Canvas, SVG, Android, iOS, …
- Supports IE8+, Chrome, Firefox, Safari
  - dropping support for IE8 in version 0.15

use "React Native"
for Android and iOS

Virtual DOM

- Secret sauce that makes React fast
- An in-memory representation of DOM
- Rendering steps
  1) create new version of virtual DOM (fast)
  2) diff that against previous virtual DOM (very fast)
  3) make minimum updates to actual DOM, only what changed
     (only slow if many changes are required)

from Pete Hunt, formerly on Instagram and Facebook React teams ...
"**Throwing out your whole UI and re-rendering it every time the data changes is normally prohibitively expensive, but with our fake DOM it's actually quite cheap.**
We can quickly diff the current state of the UI with the desired state and compute the minimal set of DOM mutations (which are quite expensive) to achieve it.
We can also batch together these mutations such that the UI is updated all at once in a single animation frame."

Client-side Model

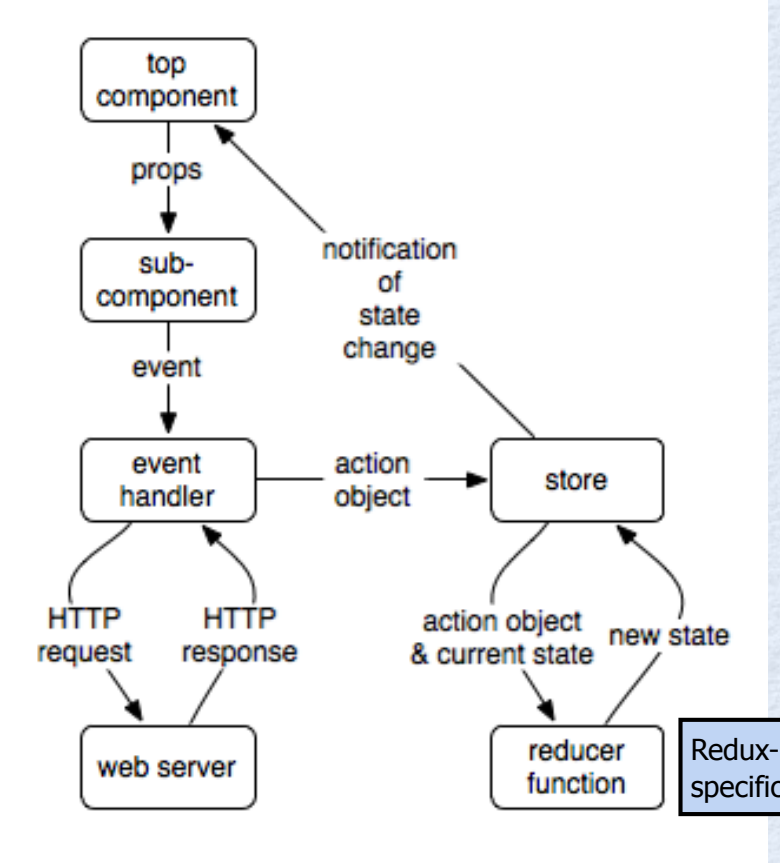
- Three options for holding client-side data ("state") used by components

- 1) Every component holds its own state
  - not recommended; harder to manage
- 2) Only a few top-level components hold state
  - these pass data to sub-components via props
- 3) "Stores" hold state
  - with **Flux** architecture there can be multiple "stores"
  - with **Redux** there is one store

My preference is to hold all state in a single, immutable object in top component that renders entire app. For an **example**, see https://github.com/mvolkmann/react-examples/tree/master/todo-reducer-rest.

Simplified Thought Process

- What DOM should each component produce with given state and props?
  - use JSX to produce DOM
- When events occur in this DOM, what should happen?
  - dispatch an action or make an Ajax call?
- Ajax calls
  - what HTTP method and URL?
  - what data to pass? pass in query string or request body?
  - update a persistent store?
  - what data will be returned in response body?
  - dispatch an action, perhaps including data from Ajax call?
- Action processing
  - how should state be updated?
- Notification of state change
  - which components need to be re-rendered?
  - just an optimization; can re-render all from top



Related Libraries

- Use other libraries for non-view functionality
- **react-bootstrap** for styling and basic widgets
  such as modal dialogs
- **Fetch** or **axios** for Ajax
- **react-router** for routing
  - maps URLs to components that should be rendered
  - supports nested views
- **Immutable** for persistent data structures with structural sharing
  - important for holding app state
  - also from Facebook - https://facebook.github.io/immutable-js/
- **Redux** for data management
  - variation on **Flux** architecture
  - uses a single store to hold all state for app
  - uses **reducer functions** that take an action and the current state, and return the new state

version of Todo app using **Redux** and **Immutable** is at https://github.com/mvolkmann/react-examples/blob/master/todo-redux-rest

Flux architecture
component -> event -> action -> dispatcher -> stores -> components

Recommended Learning Order

- From Pete Hunt
  - "You don't need to learn all of these to be productive with React."
  - "Only move to the next step if you have a problem that needs to be solved."

1. **React** itself
2. **npm** - for installing JavaScript packages
3. JavaScript bundlers - like **webpack**
4. **ES6** (ES 2015) I would do this first.
5. routing - **react-router**
6. state management with Flux - **Redux** is preferred
7. immutable state - **Immutable** library is preferred
8. Ajax alternatives - Relay (uses GraphQL), Falcor, ... Currently I would skip this.

Compared to Angular 1

Angular 1	React
module	ES6 module
directive	component (ES6 class or function)
controller	component constructor & methods
template	JSX in `render` method
service	JavaScript function
filter	JavaScript function

React feels more like writing "normal" JavaScript code

npm

- Node Package Manager
  - even though they say it isn't an acronym
- Each project/library is described by a **`package.json`** file
  - lists all dependencies (development and runtime)
  - can define scripts to be run using the "npm run" command
- To generate **`package.json`**
  - **`npm init`**
  - answer questions
- To install a package globally
  - **`npm install -g name`**
- To install a package locally and add dependency to **`package.json`**
  - for development dependencies, **`npm install --save-dev *name*`**
  - for runtime dependencies, **`npm install --save *name*`**

`package.json` Scripts

- Defined by **`scripts`** property object value
  - keys are script names
  - values are strings of shell commands to run
- Manually add script tasks
  - to do things like
    start a server,
    run a linter,
    run tests, or
    delete generated files
- To run a script, **`npm run name`**
  - can omit **`run`** keyword for special script names
- See example ahead

Special Script Names

- **`prepublish`**, **`publish`**, **`postpublish`**
- **`preinstall`**, **`install`**, **`postinstall`**
- **`preuninstall`**, **`uninstall`**, **`postuninstall`**
- **`preversion`**, **`version`**, **`postversion`**
- **`pretest`**, **`test`**, **`posttest`**
- **`prestart`**, **`start`**, **`poststart`**
- **`prestop`**, **`stop`**, **`poststop`**
- **`prestart`**, **`start`**, **`poststart`**
- **`prerestart`**, **`restart`**, **`postrestart`**

React Setup

- Install React with **`npm install --save react react-dom`**
  - **`react-dom`** is used when render target is web browsers
- Can use **`browser.js`** to compile React code in browser at runtime, but not intended for production use
- Let's start serious and use **webpack**
  - details on next slide

webpack

- https://webpack.github.io
- Module bundler
  - combines all JavaScript files starting from "entry" by following imports
  - can also bundle CSS files references through imports
- Tool automation
  - through loaders
  - ex. ESLint, Babel, Sass, ...
- Install by running **`npm install --save-dev`** on each of these:
  - babel-core, babel-loader
  - eslint, eslint-loader, eslint-plugin-react
  - webpack, webpack-dev-server

requires configuration via **`.eslintrc`**

webpack-dev-server

- HTTP server for development environment
- Provides watch and hot reloading
- Bundles are generated in memory
  and served from memory for performance

webpack.config.js

- Create **`webpack.config.js`**
  - **`entry`** is main JavaScript file that imports others
  - use babel-loader to transpile ES6 code to ES5
  - use eslint-loader to check for issues in JavaScript files
  - use css-loader to resolve URL references in CSS files
  - use style-loader to provide hot reloading of CSS
- To generate **`bundle.js`** file
  - run **`webpack`** for non-minimized
  - run **`webpack -p`** for minimized (production)

```
module.exports = {
  entry: './src/demo.js',
  output: {
    path: __dirname,
    filename: 'build/bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel!eslint'
      },
      {
        test: /\.css$/,
        exclude: /node_modules/,
        loader: 'style!css'
      },
    ]
  }
};
```

webpack.config.js

"Loading CSS requires the css-loader and the style-loader. They have two different jobs. The css-loader will go through the CSS file and find url() expressions and resolve them. The style-loader will insert the raw css into a style tag on your page."

package.json

```
{
  "name": "my-project-name",
  "version": "1.0.0",
  "description": "my project description",
  "scripts": {
    "start": "webpack-dev-server --content-base . --inline"
  },
  "author": "my name",
  "license": "my license",
  "devDependencies": {
    "babel-core": "^6.1.2",
    "babel-loader": "^6.0.1",
    "babel-preset-es2015": "^6.1.18",
    "babel-preset-react": "^6.1.2",
    "css-loader": "^0.23.1",
    "eslint": "^1.6.0",
    "eslint-loader": "^1.0.0",
    "eslint-plugin-react": "^3.5.1",
    "style-loader": "^0.13.0",
    "webpack": "^1.12.9",
    "webpack-dev-server": "^1.14.0"
   },
  "dependencies": {
    "react": "^0.14.3",
    "react-dom": "^0.14.3"
  }
}
```

to start server and watch process, enter "`npm start`"

Simplest Possible Demo

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Simplest Demo</title>
  </head>
  <body>
    <div id="content"></div>
    <script src="build/bundle.js"></script>
  </body>
</html>
```

`build/bundle.js` is generated from `src/demo.js` by webpack

`build/bundle.js` isn't actually generated when using webpack-dev-server, it's all done in memory

src/demo.js

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, World!</h1>,
  document.getElementById('content'));
```

JSX

can render into any element, and can render into more than one element

Do not render directly to document.body!
Browser plugins and other JS libraries sometimes add elements to body which can confuse React.

- Steps to run
  - `npm start`
    - assumes `package.json` configures this to start webpack-dev-server
  - browse `localhost:8080`

JSX ...

- JavaScript XML
- Inserted directly into JavaScript code
  - can also use in TypeScript
- Very similar to HTML
- Babel finds this and converts it to calls to JavaScript functions that build DOM
- Many JavaScript editors and tools support JSX
  - **editors**: Atom, Brackets, emacs, Sublime, Vim, WebStorm, ...
  - **tools**: Babel, ESLint, JSHint, Gradle, Grunt, gulp, ...

from Pete Hunt ...
"We think that **template languages are underpowered** and are bad at creating complex UIs.
Furthermore, we feel that they are **not a meaningful implementation of separation of concerns** — markup and display logic both share the same concern, so why do we introduce artificial barriers between them?"

Great article on JSX from Corey House at http://bit.ly/2001RRy

... JSX ...

- Looks like HTML, but it isn't!
  - all tags must be terminated, following XML rules
  - insert JavaScript expressions by enclosing in braces - `{ js-expression }`

    not statements!
    ex. ternary instead of `if`
  - switch back to JSX mode with a tag
  - `class` attribute -> `className`
  - `label for` attribute -> `htmlFor`

    supposedly because `class` and `for` are reserved keywords in JavaScript

    Why?
  - camel-case all attributes: ex. `autofocus` -> `autoFocus` and `onclick` -> `onClick`
  - value of event handling attributes must be a function, not a call to a function
    - use `Function bind` to specify arguments (examples later)
  - `style` attribute value must be a JavaScript object, not a CSS string
  - camel-case all CSS property names: ex. `font-size` -> `fontSize`
  - `<textarea>value</textarea>` -> `<textarea value="value"/>`
  - cannot use HTML/XML comments

    can use `{/* comment */}`

    Why?
  - HTML tags start lowercase; custom tags start uppercase

Event handling attributes in JSX are actually React-specific versions of DOM event handling. An event object is passed to the registered function, but it isn't a real DOM event. Its `e.target` refers to the React component where the event occurred. The DOM node can be obtained from that.

… JSX

- Repeated elements (ex. `li` and `tr`) require a `key` attribute
  - often an `Array` of elements to render is created using `map` and `filter` methods
  - key value must be unique within parent component
  - used in "reconciliation" process to determine whether a component needs to be re-rendered or can be discarded
  - will get warning in browser console if omitted
- Comparison to Angular
  - Angular provides custom syntax (provided directives and filters/pipes) used in HTML
  - React provides JSX used in JavaScript, a much more powerful language

Props

- JSX attributes create "props"
  - see "`name`" in next example

both standard HTML attributes and custom attributes

- Props specified on a JSX component can be accessed
  - **inside component methods** with `this.props` whose value is an object holding name/value pairs
  - **inside "functional components"** via props object that is an argument to the function
    - often ES6 destructuring is used to extract specific properties from props object

see examples of these two forms of defining components ahead

- Used to pass read-only data and functions (ex. event handling callbacks) into a component
- To pass value of a variable or JavaScript expression, enclose in braces instead of quotes
  - will see in Todo example

Reserved prop names

`dangerouslySetInnerHTML`, `key`, `ref`, any DOM properties such as `checked`, `className`, `disabled`, `href`, `htmlFor`, `id`, `name`, `on`***`EventName`***, `readonly`, `required`, `selected`, `src`, `style`, `title`, `type`, `value`, `...`

Components

- Custom components can be referenced in JSX
  - name must start uppercase to distinguish from HTML elements
- Two kinds, smart and dumb
  - **smart components** have state and/or define lifecycle methods
  - **dumb components** get all their data from props
    and can be defined in a more concise way
    ("stateless functional component" form)
    - essentially only equivalent of `render` method; no "lifecycle methods"
- Want a minimal number of smart components at top of hierarchy
- Want most components to be dumb
- Defining each component in a separate `.js` file
  allows them to be imported where needed

Component Example

src/greeting.js

```
import React from 'react';

class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

export default Greeting;
```

demonstrates using ES6 `class` syntax; can also define by calling `React.createClass`

```
import React from 'react';

export default ({name}) =>
  <h1>Hello, {name}!</h1>;
```

stateless functional component form "like a React class with only a `render` method"

props is passed and destructured

src/demo.js

```
import Greeting from './greeting';
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <Greeting name="Mark"/>,
  document.getElementById('content'));
```

must have this even though it is not directly referenced

Hello, Mark!

Events

- HTML event handling attributes (like **`onclick`**) must be camel-cased in JSX (**`onClick`**)
- Set to a function reference, not a call to a function
  - three ways to use a component method
    1. arrow function; ex. **`onClick={e => this.handleClick(e)}`**
    2. function **`bind`**; ex. **`onClick={this.handleClick.bind(this)}`**
    3. pre-bind in constructor
  - see **`onChange`** in example ahead
- Registers React-specific event handling on a DOM node
- The function is passed a React-specific event object where **`target`** property refers to React component where event occurred

more on **`bind`** and pre-bind later

best option; with other options a different value is passed as the prop value in each render which makes **`PureRenderMixin`** and **`shallowCompare`** ineffective (helpers for **`shouldComponentUpdate`**)

State

- Holds data for a component that may change over lifetime of component instance, unlike props which do not change for that component instance
  - the component may be re-rendered with different prop values
- To add/modify state properties (shallow merge), pass an object describing new state to `this.setState`
  - replaces values of specified properties and keeps others
  - triggers DOM modifications
    - unless modified state properties aren't used by the component
- To access state data, use `this.state.name`
  - example: `const foo = this.state.foo;`
  - alternative using destructuring: `const {foo} = this.state;`
- Never directly modify `this.state`
  - can cause subtle bugs

two kinds of data, app data and UI data (ex. selected sort order and filtering applied)

Function bind ...

- `bind` is a method on `Function` objects
- Creates a new function that calls an existing one
- Can do two things
  - set value of `this` inside new function
    - if it doesn't use `this`, pass `null`
  - give fixed values to initial parameters
- Choose to do one or both
- Usage

```
const newFn =
  oldFn.bind(valueOfThis, p1, p2);
```

```
function add(a, b) {
  return a + b;
}

const add5 = add.bind(null, 5);

console.log(add5(10)); // 15
```

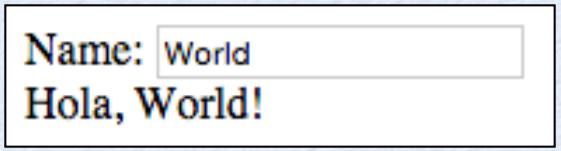
```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

  getArea() {
    return this.width * this.height;
  }
}

const r1 = new Rectangle(2, 3);
const r2 = new Rectangle(3, 4);

const getR2Area = r1.getArea.bind(r2);
console.log(getR2Area()); // 12
```

... Function bind

- Pre-binding methods from prototype of a class
  - common in React components
  - adds methods to component instance
  - if same name is used, shadows method on prototype
  - see `setName` method in next example

Event/State Example ...

- This example demonstrates an alternative to two-way data binding that is often shown in example AngularJS code



src/demo.js

```
import Greeting from './greeting';
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <Greeting greet="Hola"/>,
  document.getElementById('content'));
```

… Event/State Example

src/greeting.js

```
import React from 'react';

class Greeting extends React.Component {
  constructor() {
    super();
    this.state = {name: 'World'}; // initial state
    this.setName = this.setName.bind(this); // pre-bind
  }

  setName(event) {
    this.setState({name: event.target.value});
  }

  render() {
    return (
      <form>
        <div>
          <label>Name: </label>
          <input type="text" value={this.state.name}
            onChange={this.setName}/>
        </div>
        <div>
          {this.props.greet}, {this.state.name}!
        </div>
      </form>
    );
  }
}
```

optional **prop validation** that identifies JSX errors

```
Greeting.propTypes = {
  greet: React.PropTypes.string
};

Greeting.defaultProps = {
  greet: 'Hello'
};

export default Greeting;
```

Stateless Functional Components

- Shorter way to define a component
  - `Todo` component ahead is an example
- From https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html
  - "In idiomatic React code, **most of the components you write will be stateless, simply composing other components**."
  - "**take props** as an argument and **return** the **element** you want **to render**"
  - "behave just **like a React class with only a** `render` **method** defined."
  - "**do not have lifecycle methods**, but you can set `.propTypes` and `.defaultProps` as properties on the function"

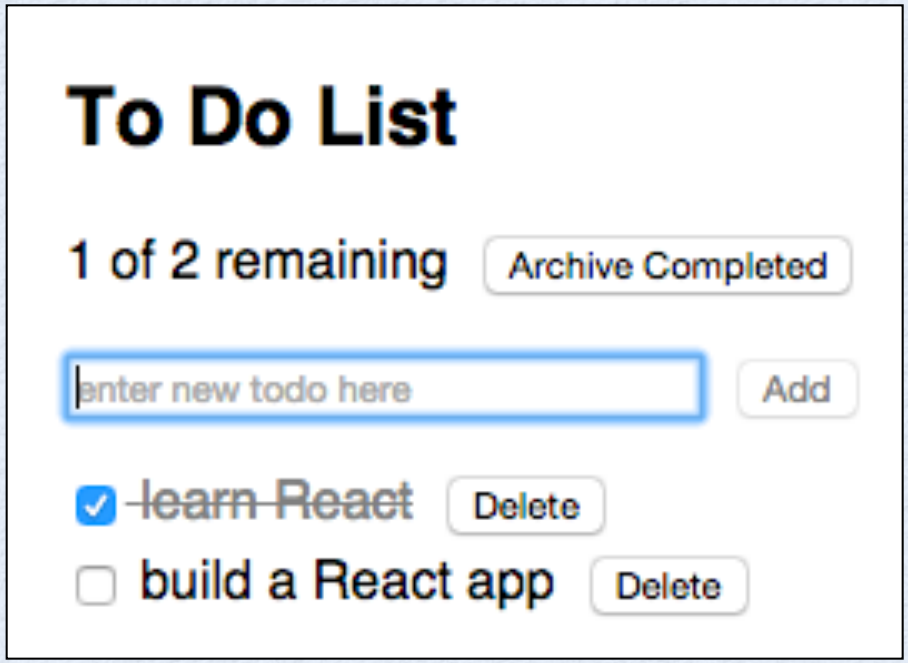
Todo List App ...

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Todo App</title>
  </head>
  <body>
    <div id="content"></div>
    <script src="build/bundle.js"></script>
  </body>
</html>
```

todo.css

```
body {
  font-family: sans-serif;
  padding-left: 10px;
}

button {
  margin-left: 10px;
}

li {
  margin-top: 5px;
}

ul.unstyled {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}

.done-true {
  color: gray;
  text-decoration: line-through;
}
```



To run:
`npm start`
browse `localhost:8080`

... Todo List App ...

todo.js

```
import React from 'react';

// props is passed to this function and destructured.
const Todo = ({todo, onToggleDone, onDeleteTodo}) =>
  <li>
    <input type="checkbox"
      checked={todo.done}
      onChange={onToggleDone}/>
    <span className={'done-' + todo.done}>{todo.text}</span>
    <button onClick={onDeleteTodo}>Delete</button>
  </li>;

const PropTypes = React.PropTypes;
Todo.propTypes = {
  todo: PropTypes.object.isRequired,
  onToggleDone: PropTypes.func.isRequired,
  onDeleteTodo: PropTypes.func.isRequired
};

export default Todo;
```

a stateless functional component

event props specify a function reference, not a call to a function

... Todo List App ...

todo-list.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import Todo from './todo';
import './todo.css';

let lastId = 0;

class TodoList extends React.Component {
  constructor() {
    super(); // must call before accessing "this"

    this.state = {
      todos: [
        TodoList.createTodo('learn React', true),
        TodoList.createTodo('build a React app')
      ]
    };

    // Pre-bind event handling methods.
    this.onArchiveCompleted = this.onArchiveCompleted.bind(this);
    this.onAddTodo = this.onAddTodo.bind(this);
    this.onTextChange = this.onTextChange.bind(this);
  }

  static createTodo(text, done = false) {
    return {id: ++lastId, text, done};
  }
```

... Todo List App ...

```
get uncompletedCount() {
  return this.state.todos.filter(t => !t.done).length;
}

onAddTodo() {
  const newTodo = TodoList.createTodo(this.state.todoText);
  this.setState({
    todoText: '',
    todos: this.state.todos.concat(newTodo)
  });
}

onArchiveCompleted() {
  this.setState({
    todos: this.state.todos.filter(t => !t.done)
  });
}
```

todo-list.js

... Todo List App ...

todo-list.js

```
onDeleteTodo(todoId) {
  this.setState({
    todos: this.state.todos.filter(t => t.id !== todoId)
  });
}

onTextChange(event) {
  this.setState({todoText: event.target.value});
}

onToggleDone(todo) {
  const id = todo.id;
  const todos = this.state.todos.map(t =>
    t.id === id ?
      {id, text: todo.text, done: !todo.done} :
      t);
  this.setState({todos});
}
```

`Array` `map` method is often used to create a collection of DOM elements from an array

Using **Immutable** would be good here because it can efficiently produce a new version of a `List` where an object at a given "key path" is updated.

... Todo List App

todo-list.js

```
  render() {
    const todos = this.state.todos.map(todo =>
      <Todo key={todo.id} todo={todo}
        onDeleteTodo={this.onDeleteTodo.bind(this, todo.id)}
        onToggleDone={this.onToggleDone.bind(this, todo)}/>);

    return (
      <div>
        <h2>To Do List</h2>
        <div>
          {this.uncompletedCount} of {this.state.todos.length} remaining
          <button onClick={this.onArchiveCompleted}>Archive Completed</button>
        </div>
        <br/>
        <form>
          <input type="text" size="30" autoFocus
            placeholder="enter new todo here"
            value={this.state.todoText}
            onChange={this.onTextChange}/>
          <button disabled={!this.state.todoText}
            onClick={this.onAddTodo}>Add</button>
        </form>
        <ul className="unstyled">{todos}</ul>
      </div>
    );
  }
}

ReactDOM.render(<TodoList/>, document.getElementById('container'));
```

can use any JavaScript to create DOM, not just a custom syntax like in templating languages or Angular

not 2-way binding

Wrapping this in a `form` causes the button to be activated when input has focus and return key is pressed.

Basic Component Definition

- **`getDefaultProps()`** not typically used
  called once regardless of # of instances created
  In components implemented with an ES6 class, set **`defaultProps`** property on class instead of implementing this method.
  - return object describing initial props for component
  - access with **`this.props`**
  - only needed for props that are not passed in by parent components
- **`getInitialState()`** not typically used
  In components implemented with an ES6 class, set **`this.state`** in **`constructor`** instead of implementing this method.
  - return object describing initial state for component
  - access with **`this.state`**
  - only needed in components that maintain their own state
-  **`render()`** must have!
  - returns component markup, typically specified with JSX
  - return **`false`** or **`null`** to render nothing
  - must be a pure function
    - return same thing for same values of **`this.state`** and **`this.props`**
    - do not modify DOM or cause other side effects
  - cannot modify **`this.state`** or **`this.props`** here

Component Life Cycle

- Three main parts
- **Mount** - initial insertion into DOM
- **Update** - re-render to virtual DOM
  to determine if actual DOM should be updated;
  triggered by state or prop changes
- **Unmount** - remove from DOM

Lifecycle Methods ...

I wish these methods did not have "component" in their name ... too verbose!

- `componentWillMount()` — rarely used — `componentWillUpdate` is a related method
  - invoked immediately before initial render
  - can create new state and pass to `this.setState` without triggering another render
- `componentDidMount()` — not typically used — `componentDidUpdate` is a related method
  - invoked immediately after initial render
  - can perform **DOM manipulation** on what was rendered
  - good place to perform setup such as loading initial data from an Ajax service and subscribing to store changes — when data is returned, pass to `this.setState`
- `componentWillReceiveProps(nextProps)` — rarely used
  - not called before initial render, but before others
  - useful for components that have state that is computed from props (not common)
  - can create new state and pass to `setState` without triggering another render

… Lifecycle Methods

- **`shouldComponentUpdate(nextProps, nextState)`**
  - not called before initial render, but before others
  - return `true` to proceed with render; `false` otherwise
  - can use to **optimize performance** by avoiding unnecessary virtual DOM creation, diffing, and re-rendering
- **`componentWillUpdate(nextProps, nextState)`**
  - not called before initial render, but before others
  - cannot call `setState` here
  - for performing "preparation" before render
- **`componentDidUpdate(prevProps, prevState)`**
  - called after updates are flushed to DOM, but not after initial render
  - can perform **DOM manipulation** on what was rendered
- **`componentWillUnmount()`**
  - called immediately before a component is removed from DOM

can efficiently compare old and new state and prop values if they are held in immutable objects

functional components can't do this; returning `null` causes nothing to be rendered which is different than avoiding re-rendering

rarely used

not typically used

not typically used

Order of Invocation

- Mount (initial render)
  - `getDefaultProps`
  - `getInitialState`
  - `componentWillMount`
  - `render`
  - `componentDidMount`
- Unmount
  - `componentWillUnmount`
- Property Change
  - `componentWillReceiveProps`
  - `shouldComponentUpdate`
  - `componentWillUpdate`
  - `render`
  - `componentDidUpdate`
- State Change
  - `shouldComponentUpdate`
  - `componentWillUpdate`
  - `render`
  - `componentDidUpdate`

Biggest Issues

- Must choose a way to efficiently modify state
  - **Immutable** library from Facebook is a good choice, but there are other options
- Constant need to use `Function` `bind` for event handlers
  - somewhat better with helper functions
- JSX is like HTML, but it's not
  - it seems there could be fewer differences
- Cannot use external HTML files
  - must specify DOM in JavaScript, typically using JSX
- No help with form validation
  - but there are third party solutions
  - doing this in plain JavaScript isn't so bad

Biggest Benefits

- Easier to create custom React components than to create Angular directives
- Fast due to use of virtual DOM and DOM diffing
- One way data flow makes it easier to understand and test components
- Can use same approach for rendering to DOM, Canvas, SVG, Android, iOS, ...

Big Questions

- Is it easier to learn and use React
  than AngularJS?
- Should my team use React
  in a new, small project
  to determine if it is a good fit for us?

The End

- Thanks so much for attending my talk!
- Feel free to find me later and ask questions about React or anything in the JavaScript world
- **Contact me**

Mark Volkmann, Object Computing, Inc.

Email: mark@ociweb.com

Twitter: @mark_volkmann

GitHub: mvolkmann

Website: http://ociweb.com/mark