React

Overview ...

- Web app library from Facebook
  - http://reactjs.org/
- 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
    - events -> state changes -> component rendering

UI as a function of app state

As of 8/6/16, **React** was reportedly **used by** Airbnb, Angie’s List, Atlasssian, BBC, Capitol One, Clash of Clans, Codecademy, Coursera, Docker, Dropbox, Expedia, **Facebook**, Feedly, Flipboard, HipChat, IMDb, **Instagram**, Intuit, Khan Academy, Lyft, New York Times, NFL, NHL, **Netflix**, **Paypal**, Periscope, Reddit, Salesforce, Squarespace, Tesla Motors, **Twitter**, Uber, Visa, WhatsApp, Wired, Wolfrum Alpha, Wordpress, Yahoo, Zendesk, and many more.
Source: https://github.com/facebook/react/wiki/Sites-Using-React

... Overview

- Defines components that are composable
  - whole app can be one component that is built on others
- Components get data to render from “props” and/or “state”
- 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, ...

use “React Native”
for Android and iOS

- 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
- Supports IE9, Chrome, Firefox, Safari

ThoughtWorks Tech Radar 4/16

● ADOPT ?

82. ES6
83. React.js
84. Spring Boot
85. Swift

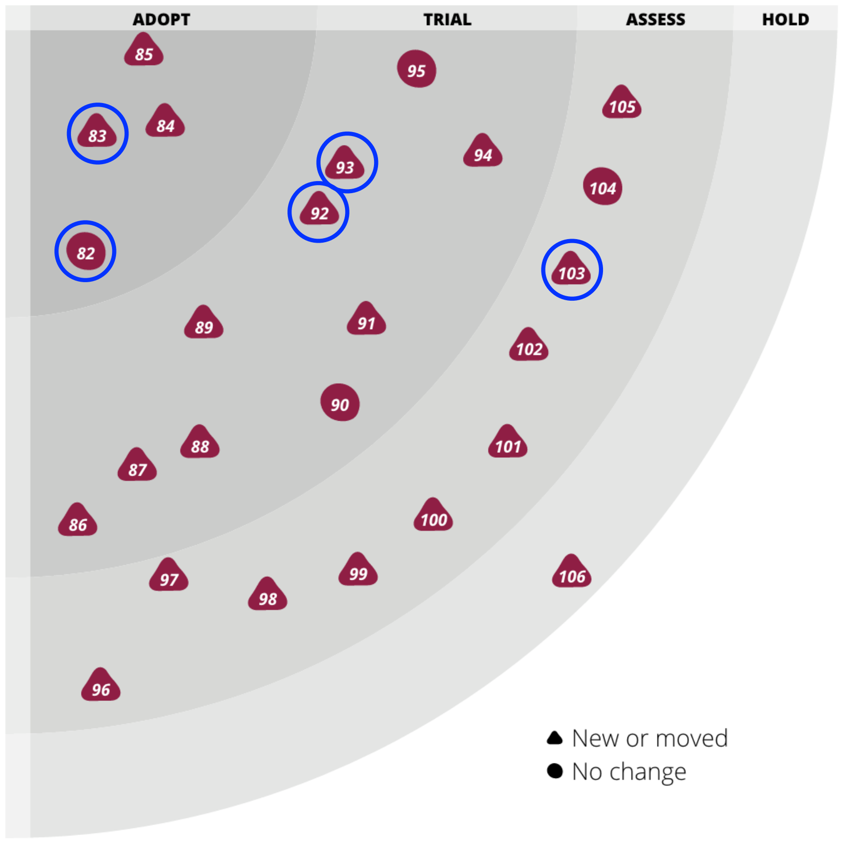
● TRIAL ?

86. Butterknife new
87. Dagger new
88. Dapper new
89. Ember.js
90. Enlive
91. Fetch new
92. React Native
93. Redux new
94. Robolectric new
95. SignalR

● ASSESS ?

96. Alamofire new
97. AngularJS — demoted from TRIAL
98. Aurelia new
99. Cylon.js new
100. Elixir
101. Elm
102. GraphQL new
103. Immutable.js new
104. OkHttp
105. Recharts new

We "have certainly seen codebases become overly complex from a combination of two-way binding and inconsistent state-management patterns."



ThoughtWorks Quotes

- In the avalanche of front-end JavaScript frameworks, **React.js stands out** ...
    - due to its design around a reactive data flow. Allowing only **one-way data binding greatly simplifies the rendering logic** and avoids many of the issues that commonly plague applications written with other frameworks. We're seeing the benefits of React.js on a growing number of projects, large and small, while at the same time we continue to be concerned about the state and the future of other popular frameworks like AngularJS. This has led to React.js becoming **our default choice for JavaScript frameworks**.
- **Redux is a great, mature tool** that has helped many of our teams
    - reframe how they think about **managing state in client-side apps**. Using a Flux-style approach, it enables a loosely coupled state-machine architecture that's **easy to reason about**. We've found it a good companion to some of our favored JavaScript frameworks, such as Ember and React.
- **Immutability** is often emphasized in the functional programming paradigm,
    - and most languages have the ability to create immutable objects, which cannot be changed once created. **Immutable.js** is a library for JavaScript that **provides many persistent immutable data structures, which are highly efficient** on modern JavaScript virtual machines. ... Our teams have had value using this library for tracking mutation and maintaining state, and it is a library we encourage developers to investigate, especially when it's combined with the rest of the Facebook stack.

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

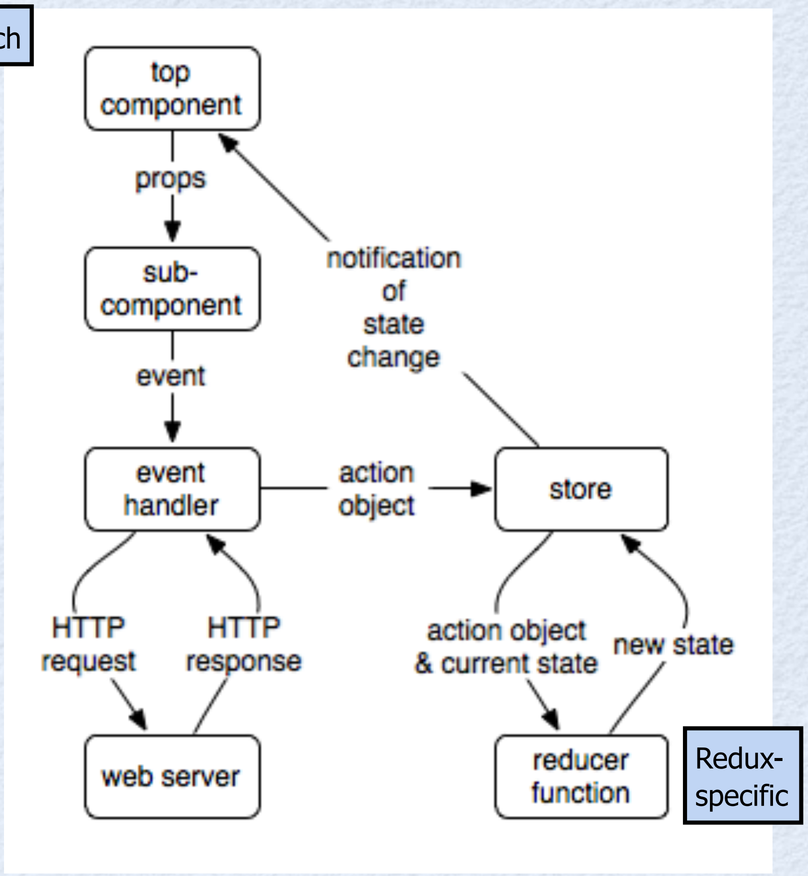
- Three options for holding client-side data (“state”) used by components

- 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

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

assuming Flux approach



Recommended Steps

- Summary of “Thinking in React”
  - https://facebook.github.io/react/docs/thinking-in-react.html
- Divide UI into a component hierarchy
- Build static version of app
  - components with no event handling
  - won’t be able to access all views
- Identify minimal UI state representation
  - everything that cannot be computed
- Identify where each piece of state data should live
  - which component owns it; maybe use a Flux store
- Add event handling

Related Libraries

- Use other libraries for non-view functionality
- **react-bootstrap** for styling and basic widgets

such as modal dialogs

also consider Material Design and Foundation

- **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/

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

- **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

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** supports use of ES6 modules
4. **ES6** (ES 2015)

---

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

React feels more like writing "normal" JavaScript code

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 → `<h1>Hello, World!</h1>`

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`

HTML in My JS?

- Yes, but ...
- Typically React apps have three primary kinds of JS files
  - component definition
  - event handling
  - state management (ex. Redux reducer functions)
- HTML only appears in JS files that define components
- Every line in those files is focused on deciding what to render
- So HTML is not out of place there ... same concern

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 }`
  - switch back to JSX mode with a tag
  - `class` attribute -> `className`
  - `label for` attribute -> `htmlFor`
  - 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
  - `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
  - HTML adds a space between adjacent elements; JSX doesn't; use `{' '}` to get this
  - HTML tags start lowercase; custom tags start uppercase

not statements!
ex. ternary instead of `if`

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

Why?

can use `{/* comment */}`

Why?

... 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

Reserved prop names

`dangerouslySetInnerHTML`, **`children`**, **`key`**, and **`ref`**

- 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 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

Components

- Custom components can be referenced in JSX
  - names 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

react-examples/component

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"

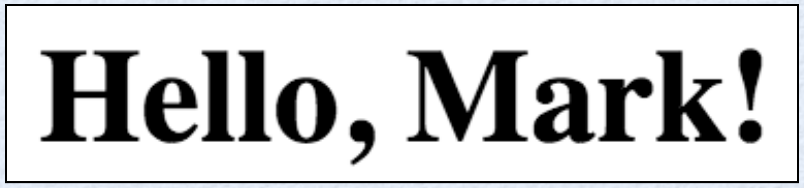
props is passed and destructured

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!

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”

Lab

- Modify code in react-examples/component
- Change **`Greeting`** component to support a **`shout`** prop
- Hints
  - can use ES6 class or stateless functional component form
  - set a variable to message to be displayed
  - declare variables with **`let`** or **`const`** instead of **`var`**
  - use a template literal to build message;
    ex. `` `prefix${variable-or-expression}suffix` ``
  - if **`shout`** prop is true, change message to all uppercase
  - use JavaScript **`String`** method **`toUpperCase`**
  - if using stateless functional component form, remember that arrow function bodies must be in braces if more than one line
  - add use of **`shout`** prop in **`demo.js`** - **`shout={true}`**
- See steps in README.md to build and run

Form Elements

- **`input`**, **`textarea`**, and **`select`** elements can have their value specified in two ways
  - controlled and uncontrolled
  - in both cases the user can change the value

Controlled Components

- Preferred over uncontrolled
  - easier to keep UI in sync with form element values
  - ex. disabling a button when an input has no value
- Use **`value`** attribute set to a state property

  or a prop derived from state

  - ex. **`<input value={this.state.`*`some-property`*`} .../>`**
- Use event handling attribute like **`onChange`** to update state properties by calling **`this.setState`** when user changes value
  - changes element value and re-renders
  - good because it keeps DOM value and state in sync

Uncontrolled Components

- Use **`defaultValue`** attribute to specify initial rendered value
  - ex. **`<input defaultValue={`*`some-expression`*`} .../>`**
  - **`input`** elements with **`type checkbox`** or **`radio`** can use **`defaultChecked`** attribute
- Use a **`ref`** to access corresponding DOM element and get/set DOM properties like value
  - but setting doesn't trigger a re-render like calling **`this.setState`**

refs are covered on next slide

Refs

- Simplify accessing rendered DOM nodes

```
// In render method ...
<input ref="myInput" ...>

// In other methods ...
const myInput = this.refs.myInput;
```

- Can get and set DOM properties `this.refs.myInput.value = '';`
- Can call DOM methods `this.refs.myInput.focus();`
- Another way to access a DOM node
  - `ReactDOM.findDOMNode(componentRef);` Use of this is discouraged!
  - can use `this` keyword inside a component method to get ***`componentRef`***
- For more information
  - see https://facebook.github.io/react/docs/more-about-refs.html
  - especially see "Cautions" section at bottom

Child Elements

- Custom components can decide where and how to render their child components
- Children are passed to parent component in **`props.children`**
  - multiple children -> array
  - single child -> element
  - no children -> **`undefined`**
- Render with **`{props.children}`**
- Similar to AngularJS 1 transclusion

Component Size

- In general, components that render large amounts of JSX should be split into smaller components
- Original component renders the new, smaller components
- Small components are easier to use, modify, and test

Component Types

- “Presentational components”
  - render DOM elements
  - respond to events by calling functions supplied via props or state
  - do not dispatch actions
- “Container components”
  - render presentational components
  - pass functions to them as props
  - these functions dispatch actions to store(s)
- Goal of separating these is to
  separate UI logic from event handling logic
  - only do this if it simplifies components

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, pass an object describing new state to `this.setState`
  - replaces values of specified properties and keeps others
  - performs a shallow merge
  - can also pass a function that returns this object
    - it is passed two objects, one containing current state and one containing current props
  - second, optional argument is a callback function that is invoked after state is modified and component re-renders
  - triggers DOM modifications
    - unless modified state properties aren't used by the component
- `this.replaceState` is similar, but replaces all existing state properties

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

... State

- `setState` and `replaceState` do not immediately mutate state
  - they create a pending state transition
- 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

Where To Hold Data

- Options are **prop**, **state**, or **store** (Flux)
- If the component won't change it in one of its own methods, use a **prop**
    - parent component can re-render this component with a new value for the prop
    - preferred location for most data
- If changes need to be persisted when component is unmounted and restored when component is remounted, use a **store**
- If an unrelated component needs to change it, use a **store**
    - here unrelated means not an ancestor or descendant
- Otherwise use **state**
- **state** is for data that is "owned" by the component

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
  - done in constructor
  - adds methods to a class that override methods on prototype
  - if same name is used, shadows method on prototype
  - see `setName` method in next example
  - using `bind` in `render` method works, but slows rendering

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
  - four 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
    4. write handler as a "public class field"
  - 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

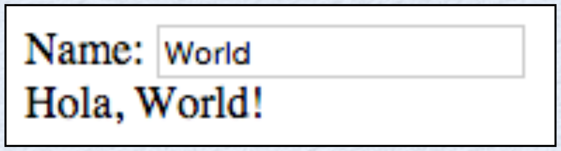
With **Redux** there is no need to use `this` in event handling methods, so `bind` isn't needed!

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/Event Example ...

react-examples/events

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



Hola, World!

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

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

src/demo.js

defined on next slide

assumes same HTML as on slide 13

... State/Event Example

src/greeting.js

```
import React from 'react';

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

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

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

export default Greeting;
```

`constructor` can take `props` parameter in order to initialize `state` from `props` values; must pass `props` to `super()`

Prop Validation ...

- Optional, but highly recommended to find JSX errors faster
- Not performed in production builds
- **`npm install prop-types`**
- Specified via component **`propTypes`**
  - an object where keys are property names and values are validation specifications
  - defined by properties on **`React.PropTypes`**

```
MyComponent.propTypes = { ... };
```

- Example

```
import PropTypes from 'prop-types';
const {func, object} = PropTypes;
Todo.propTypes = {
  todo: object.isRequired,
  onToggleDone: func.isRequired,
  onDeleteTodo: func.isRequired
};
```

alternative way to define

```
import PropTypes from 'prop-types';
const {func, object} = PropTypes;

// inside class definition
static propTypes = {
  todo: object.isRequired,
  onToggleDone: func.isRequired,
  onDeleteTodo: func.isRequired
};
```

... Prop Validation

- Validation options
  - primitive types: **`bool`**, **`number`**, **`string`**
  - function: **`func`**
  - DOM types: **`element`**, **`node`**
  - enums: **`oneOf`**, **`oneOfType`**
  - arrays: **`array`**, **`arrayOf`**
  - objects: **`object`**, **`objectOf`**, **`instanceOf`**, **`shape`**
  - custom: a function that takes **`props`**, **`propName`**, and **`componentName`**
    - useful for complex validation such as evaluating values of other properties
    - access value to be validated with **`props[propName]`**
    - return an **`Error`** object if validation fails; nothing otherwise
  - any type: **`any`**

`oneOf` specifies an array of allowed literal values

`oneOfType` specifies an array of validation options

`shape` specifies properties that must be present in an object, and their types (see example later)

only useful when type doesn't matter, but prop must be present

- Props are optional by default
  - add **`.isRequired`** at end of validation option to make required

Todo List App ...

react-examples/todo

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

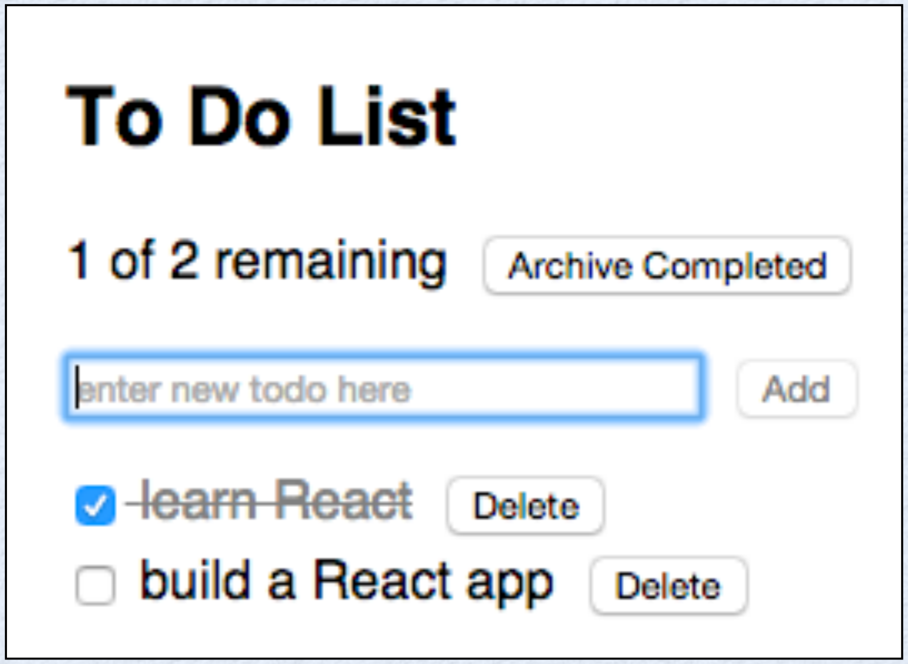
```
body {
  font-family: sans-serif;
  padding-left: 10px;
}

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

export default Todo;
```

a stateless functional component

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

Validating todo prop using shape

```
const {bool, func, shape, string} = React.PropTypes;
Todo.propTypes = {
  todo: shape({
    done: bool.isRequired,
    text: string.isRequired
  }).isRequired,
  onDeleteTodo: func.isRequired,
  onToggleDone: func.isRequired
};
```

... 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 = {
      todoText: '', // must initialize
      todos: [
        TodoList.createTodo('learn React', true),
        TodoList.createTodo('build a React app')
      ]
    };

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

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

... Todo List App ...

todo-list.js

```
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 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});
}
```

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('content'));
```

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

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

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

not 2-way binding

Object Spread

- Upcoming JavaScript feature that is supported by Babel
- Provides another way to pass props to a component that can be more concise
- Can come from any object, including **`this.state`** and **`this.props`**

```
// Without object spread
<Todo key={todo.id} todo={todo}
  onDeleteTodo={this.onDeleteTodo.bind(this, todo.id)}
  onToggleDone={this.onToggleDone.bind(this, todo)}/>);

// With object spread
const todoProps = {
  key: todo.id,
  todo,
  onDeleteTodo: this.onDeleteTodo.bind(this, todo.id),
  onToggleDone: this.onToggleDone.bind(this, todo)
};
<Todo {...todoProps}/>
```

- Named properties override those from object spread

```
<MyComponent {...someObject} luckyNumber=7 prize={myPrize}>
  ...
</MyComponent>
```

Setting CSS Properties

- Create an object where keys are camel-cased CSS property names
- Use as value of **`style`** attribute

```
// Suppose state holds a color.
render() {
  const style = {
    backgroundColor: this.state.bgColor,
    fontSize: '24pt'
  };

  return (
    <div style={style}>
      ...
    </div>
  );
}
```

Immutability

- Mutating **`this.state`** is taboo!
- Making deep copies can be expensive
- Options are discussed in Immutability section

Public Class Fields

- A stage 2 TC39 proposal as of 1/6/17
- Supported by Babel
- Can eliminate need for component constructors and using `bind` with event handling methods

```
class MyComponent extends React.Component {

  // Can define state as a public field
  // instead of inside constructor.
  state = {
    ...
  };

  // Can define event handling functions
  // as public fields instead of as a method
  // which removes the need to use bind.
  onSomething = () => {
    ...
  };

  ...
}
```

- Explained by Kent C. Dodds in an Egghead video here
  - https://egghead.io/lessons/javascript-public-class-fields-with-react-components

Biggest Issues

- For large apps, need to choose a way to efficiently modify state
  - **Immutable** library from Facebook is a good choice
- Often need to use **`Function`** **`bind`** for event handlers
  - not needed when a Flux library is used
- Cannot use external HTML files
  - must specify DOM in JavaScript, typically using JSX
- JSX is like HTML, but it’s not
  - it seems there could be fewer differences

Biggest Benefits

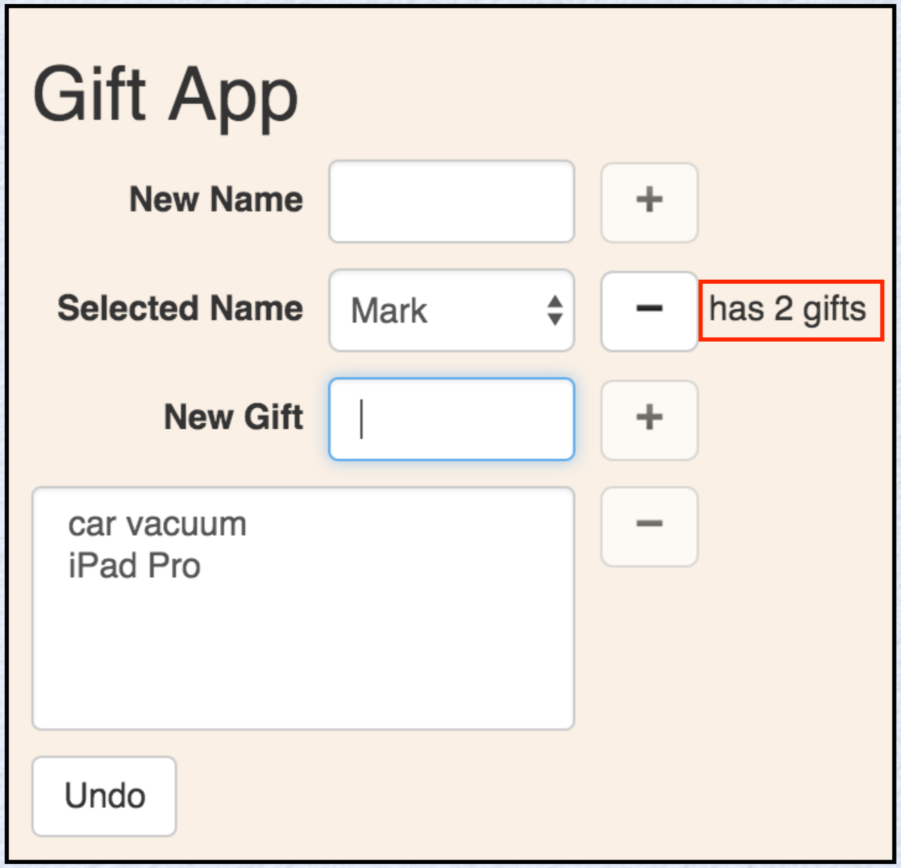
- Emphasizes using JavaScript rather than custom template syntax to build views
  - ex. JavaScript `if` and ternary operator versus `ng-if`, `ng-show`, and `ng-hide`
  - ex. JavaScript `Array` `map` method versus `ng-repeat`

comparing to Angular 1

- Easier to create custom React components than to create Angular directives
  - just need a `render` method or stateless functional component
- Fast due to use of virtual DOM and DOM diffing
- One way data flow makes it easier to understand and test components
  - most components only use data that is directly passed to them via props
- Very easy to write component tests
- Can use same approach for rendering to DOM, Canvas, SVG, Android, iOS, ...

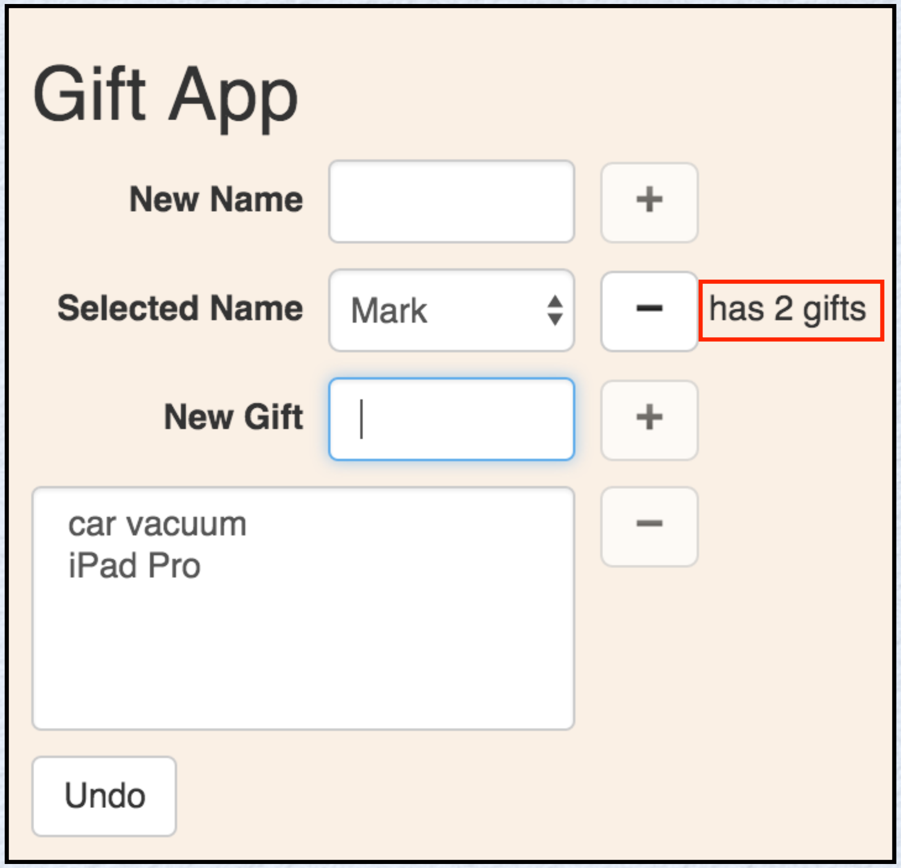
Lab …

- cd to react-examples/gift
- Follow steps in **`README.md`** to build and run
- We will discuss the functionality of this app and review the code
  - **`index.html`**
  - **`main.js`**
  - **`app.scss`**
  - **`gift-app.js`**
  - **`text-entry.js`**
  - **`name-select.js`**
  - **`gift-list.js`**
  - **`autobind.js`**
  - **`deep-equal.js`**



... Lab

- Add display of number of gifts for selected person (part in red)
- Hints
  - modify **`name-select.js`** **`render`** method
    - add use of **`giftCount`** prop (a number) to **`NameSelect`** component
    - set a variable to message to be displayed
    - render message after button
  - modify **`gift-app.js`** **`render`** method
    - add **`giftCount`** prop to use of **`NameSelect`** component
    - **`giftCount={giftsForName.length}`**


Gift App
New Name
+
Selected Name
Mark
−
has 2 gifts
New Gift
+
car vacuum
iPad Pro
−
Undo