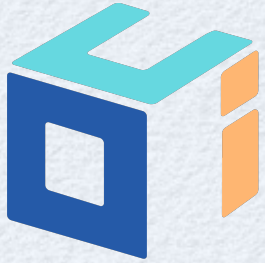
# Full Stack JS
# Client Side

**R. Mark Volkmann**
Object Computing, Inc.
http://objectcomputing.com
Email: mark@objectcomputing.com
Twitter: @mark_volkmann
GitHub: mvolkmann



OCI | TRAINING

# Outline

- React overview
- create-react-app
- Components
- JSX
- Props
- Prop Validation
- State
- Events
- Form Elements
- Basic Todo App
- Lifecycle Methods
- Managing State
- Redux
- redux-easy
- Todo App using redux-easy
- REST calls
- Routing
- Jest test framework
- react-testing-library
- Hands on exercise

**Preparation Steps**

```
git clone https://github.com/mvolkmann/react-tour-of-heroes
cd react-tour-of-heroes
cd server
npm install
cd ../client
npm install
```

# React Overview ...

- Web app library from Facebook
  - http://facebook.github.io/react/
- Focuses on view portion
  - not full stack like other frameworks such as Angular
  - use other libraries for non-view functionality
    - some are listed later
- "One-way reactive data flow"
  - UI reacts to "state" changes
  - easier to follow flow of data
    - events -> state changes -> component rendering
  - makes UI a function of app state
    - like "pure functions" in functional programming

# … React Overview

- Defines components that are composable
  - whole app can be one component that is built on others
- Components get data 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
- 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
  - can create custom HTML elements that use a
    "Web Component" to render a React component
    - see https://github.com/mvolkmann/talks/blob/master/ReactInAngular.key.pdf
- Supports Chrome, Firefox, IE9+, and Safari

# Virtual DOM

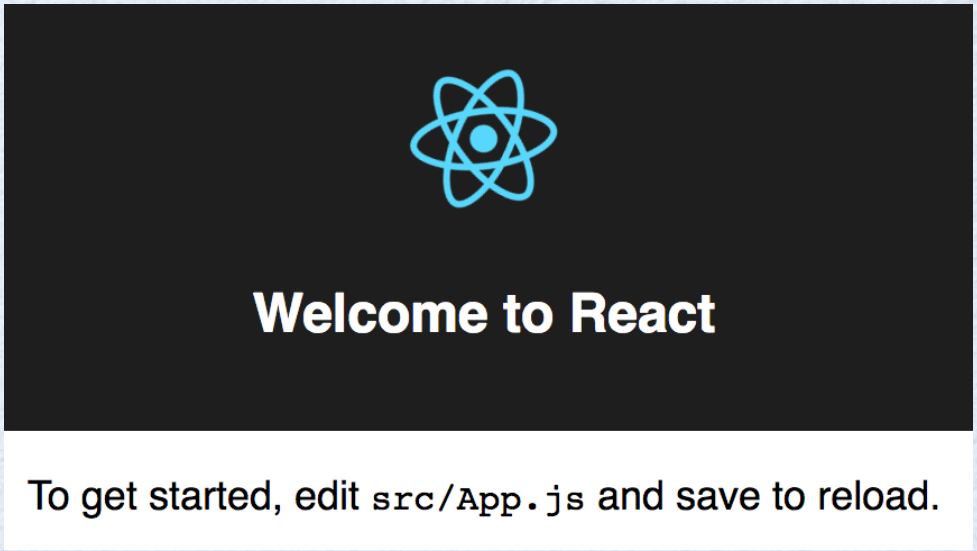
- Secret sauce that makes React fast
- An in-memory representation of DOM
- Rendering steps
  1) create new version of virtual DOM (fast)
  2) diff that against previous virtual DOM (very fast)
  3) make minimum updates to actual DOM, only what changed
     (only slow if many changes are required)

**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."

# create-react-app

https://github.com/facebook/create-react-app

- Tool that creates a great starting point for new React apps
- **`npm install -g create-react-app`**
- **`create-react-app`** ***`app-name`***
  - takes about 20 seconds to complete because it downloads and installs many npm packages
- **`cd`** ***`app-name`***
- **`npm start`**
  - starts local HTTP server
  - opens default browser to local app URL
- Don't eject!

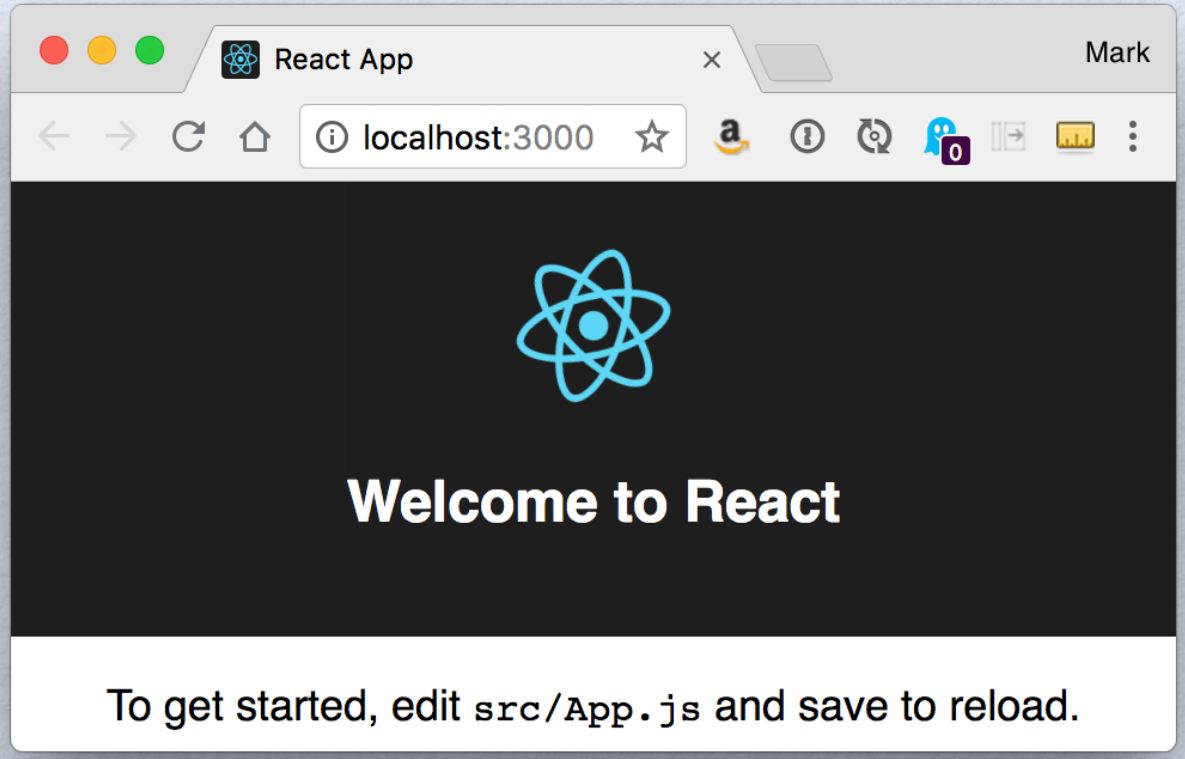


# Benefits of create-react-app

- Creates directory structure and files including `package.json`
- Installs and configures many tools and libraries
- Provides a **local web server** for use in development
- Provides **watch and live reload**
- Uses **Jest** test framework which supports **snapshot tests**
- Lets Facebook maintain the build process
  - benefit from future improvements
- Produces small production deploys

# Try It!

- Create a new app using create-react-app
  - `npm install -g create-react-app`
  - `create-react-app midwest-js-app`
    - requires internet connection
  - `cd midwest-js-app`
- Start it
  - `npm start`
- We'll modify this later



# Examine It!

- **public/index.html**
  - contains a `<div>` with an `id` of "`root`"
  - React will render a component here
- **src/index.js**
  - important lines are

```
import App from './App';

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

- **src/app.js**
  - defines the `App` component using a class that extends `Component`
  - has a `render` method

```
import React, {Component} from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo}
            className="App-logo"
            alt="logo" />
          <h1 className="App-title">
            Welcome to React
          </h1>
        </header>
        <p className="App-intro">
          To get started, edit
          <code>src/App.js</code>
          and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

This is **JSX**. We'll talk about this more soon.

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

# How Does It Work?

- For development,
  "**`npm start`**" uses webpack-dev-server to serve the app
  - configured to use **`src/index.js`** as the "entry point"
  - so **`index.html`** doesn't need a **`script`** tag
- For production,
  "**`npm build`**" uses webpack to
  compile all the assets and produce a "bundle"
  - bundle is stored in **`build/static/main.*.js`**
  - **`html-webpack-plugin`** injects a **`script`** tag into **`build/index.html`**
    that refers to the bundle

# Modify It!

- Edit **`src/App.js`**
  - change "**`Welcome to React`**"
    to "**`Welcome to Midwest JS`**"
- Save the change and note that
  the browser is automatically updated!

# Components

- Two ways to implement
  - plain JavaScript function
  - class that extends `React.Component`
    - useful when event handling and/or lifecycle methods are required
- Specify what to render, typically using JSX which is very similar to HTML
  - described more later
- Defining each component in a separate file allows them to be imported where needed
- To render a component, import it in the definition of another component and return JSX for it
  - example

```
// Add this near top of App.js.
import Demo from './demo';

// Add this in render method of App.js.
<Demo />
```

# Component Functions

```
import React, {Component} from 'react';

const onClick = () => alert('got click');

export default () => (
  <div className="demo">
    <button onClick={onClick}>Press Me</button>
  </div>
);
```

demo.js

specifying **className** makes it easy to style

- Returns same as **`render`** method in class form on next slide

# Component Classes

demo.js

```
import React, {Component} from 'react';

class Demo extends Component {
  onClick = () => alert('got click');

  render() {
    return (
      <div className="demo">
        <button onClick={this.onClick}>Press Me</button>
      </div>
    );
  }
}

export default Demo;
```

doesn't need to be a method unless it uses `this`

"class public field" TC39 stage 2 proposal supported by Babel and create-react-app

- Use this form when any of the following are needed
  - component state
  - instance or static properties
  - instance or static methods
  - lifecycle methods (such as `componentDidMount`)

more on these later

# HTML in My JS?

- Technically it is JSX
- Initially this feels wrong to most developers
- But using the full power of JavaScript
  to dynamically generate the DOM
  using conditional logic and iteration
  is a good thing!
  - many would say better than inventing a mini-language
    that is inserted into HTML as is done in Angular and Vue

# JSX

- JavaScript XML
- Inserted directly into JavaScript or TypeScript code
- Very similar to HTML
- Babel finds this and converts it to calls to JavaScript functions that typically build DOM
- Many JavaScript editors and tools support JSX and let you know when there are mistakes
  - **editors**: Atom, emacs, Sublime, Vim, VS Code, 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 Differences from HTML

- HTML tags start lowercase; custom tags start uppercase
- 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
- Cannot use HTML/XML comments
  - can use JavaScript comments with `{/* comment */}`
- `class` attribute -> `className`
  - because `class` is a reserved JavaScript keyword
- 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
- and a few more that don't come up often

# Props

- Primary way to pass read-only data and functions to components
  - functions can be used as callbacks and "render props"
- JSX attributes create "props"
- Props can be accessed
  - **inside function components** via props object argument to the function
  - **inside class component methods** with `this.props`
  - either way the value is **an object holding name/value pairs**
  - often **destructuring** is used to extract specific properties from the props object
- To pass value of a variable or JavaScript expression, enclose in braces instead of quotes
  - ex. `<Greeting name={name} />`

# Component Using Prop

src/greeting.js

```
import React from 'react';

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

export default Greeting;
```

```
import React from 'react';

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

props is passed and destructured

src/App.js

```
import Greeting from './greeting';

<Greeting name="Mark" />,
```

**Hello, Mark!**

# Try It!

- Add a component named **`Calculate`**
  - create **`calculate.js`** in the **`src`** directory
  - implement a component that takes the following props:
    - **`numbers`** - an array of numbers
    - **`operation`** - the string **`"add"`** or **`"multiply"`**
  - compute the result from these props →
  - render result

```
return <div>{result}</div>;
```

```
const {numbers, operation} = this.props;
let result = 0;
switch (operation) {
  case 'add':
    result = numbers.reduce(
      (sum, n) => sum + n);
    break;
  case 'multiply':
    result = numbers.reduce(
      (product, n) => product * n, 1);
    break;
  default:
}
```

- Render your component from **`App.js`**

```
import Calculate from './calculate';
const values = [1, 3, 7];
...
<Calculate numbers={values} operation="add" />
```

# Prop Validation ...

Can use Flow or TypeScript instead for even more type checking!

- Optional, but highly recommended to find errors faster
  - displayed in browser console
- Not performed in production builds
- To use

```
import {PropTypes} from 'prop-types';
```

installed by create-react-app

- For function components

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

  - **propTypes** is an object where keys are property names and values are validation specifications
- For class components

```
// inside class definition
static propTypes = { ... };
static defaultProps = { ... };
```

Example

```
const {func, object} = PropTypes;
Todo.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`**

    **`oneOf`** specifies an array of allowed literal values

    **`oneOfType`** specifies an array of validation options
  - arrays: **`array`**, **`arrayOf`**
  - objects: **`object`**, **`objectOf`**, **`instanceOf`**, **`shape`**

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

    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

for more details on prop validation, see https://github.com/facebook/prop-types

# State

Most components only get data from props, like "pure functions".

- Holds data for a component instance that may change over its lifetime
- Declare initial state at top of class definition `state = {age: 0, name: ''};`
- To add/modify state properties, call **`this.setState`**
  - **approach #1**: pass an object describing state changes
    - replaces values of specified properties and keeps others
    - performs a shallow merge

```
this.setState({
  name: 'Tami'
});
```

  - **approach #2**: pass a function
    - passed current state and returns an object describing state changes
    - **use when changes are based on current state**

```
this.setState(state => ({
  age: state.age + 1
}));
```

  - both approaches can trigger DOM modifications
- To access state data, use **`this.state.`*`name`***
  - ex. `const {foo} = this.state;`
- Never directly modify **`this.state`**
  - can cause subtle bugs
  - see https://reactjs.org/docs/react-component.html#setState

# Try It!

- Modify `Calculate` component so it holds the result of the operation in its own state rather than recomputing in each render
  - requires a class-based component, so if it is currently function-based, change it
  - at top of class definition, add `state = {result: 0};`
  - define the `getDerivedStateFromProps` lifecycle method which is invoked when the component is first rendered and again every time different prop values are passed in

    lifecycle methods are discussed more later

```
static getDerivedStateFromProps(nextProps, prevState) {
  const {numbers, operation} = nextProps;
  // Calculate result from numbers and operation.
  return {...prevState, result};
}
```

  - render `this.state.result`

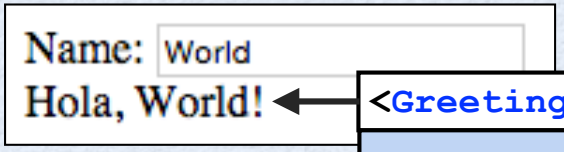
```
render() {
  return <div>{this.state.result}</div>;
}
```

# 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
  - there are many options for using a component method instead of a plain function
  - best options are
    1. use a public class field see `setName` example ahead
    2. use an arrow function; ex. `onClick={event => this.handleClick(event)}`
- Passed a React-specific event object
  - `target` property refers to React component where event occurred

# State/Event Example ...

react-examples/event-demo




```
<Greeting greet="Hola" />,
```

defined on next slide

# … State/Event Example

src/greeting.js

optional **prop validation**

```
import React, {Component} from 'react';
import {string} from 'prop-types';

class Greeting extends Component {
  static propTypes = {greet: string};
  static defaultProps = {greet: 'Hello'};

  state = {name: 'World'}; // initial state

  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>
    );
  }
}

export default Greeting;
```

# Try It!

- Add a "Bigger" button to the `Calculate` component that increases the font size
- Add `fontSize` property to state

```
state = {fontSize: 10, result: 0};
```

- Add `bigger` method using a class public field

```
bigger = () => this.setState(state => ({fontSize: state.fontSize + 2}));
```

- Add a button and use of `style` prop to `render` method

```
render() {
  const style = {fontSize: this.state.fontSize};
  return (
    <div style={style}>
      {this.state.result}
      <button onClick={this.bigger} style={style}>
        Bigger
      </button>
    </div>
  );
}
```

# Form Elements

- **`input`**, **`textarea`**, and **`select`** elements can have their value specified in two ways
  - controlled and uncontrolled
  - in both cases users can change the value
- **Controlled components** - preferred
  - specify current value with **`value`** prop
  - specify event handling props (like **`onChange`**) to respond to changes

```
<input
  onChange={this.handleChange}
  value={this.props.lastName}
/>
```

- **Uncontrolled components** - not preferred
  - can specify **`defaultValue`** or **`defaultChecked`** prop to provide initial value
  - do not specify current value and instead rely on the DOM to hold it
  - can use a **`ref`** to access corresponding DOM element and get/set DOM properties like **`value`**
    - not covered here

# 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

# Child Elements

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

my-container.js

```
import React from 'react';

export default ({children}) => (
  <div>
    <div>Before Children</div>
    {children}
    <div>After Children</div>
  </div>
);
```

App.js

```
import MyContainer from './my-container';
...
<MyContainer>
  <div>child #1</div>
  <div>child #2</div>
</MyContainer>
```

Before Children
child #1
child #2
After Children

# Object Spread

a feature added in ES2018

- Provides another way to pass props to a component that can be more concise
- Can come from any object

```
// Without object spread
<MyComponent foo={true} bar="1" baz="qux" />

// With object spread
const data = {foo: true, bar: 1, baz: "qux"};
<MyComponent {...data} />
```

- Named properties override those from object spread

```
<MyComponent {...data} bar="2" />
```

# Todo List App ...

react-examples/todo

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import TodoList from './todo-list';

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

modified from what create-react-app generated

todo.css

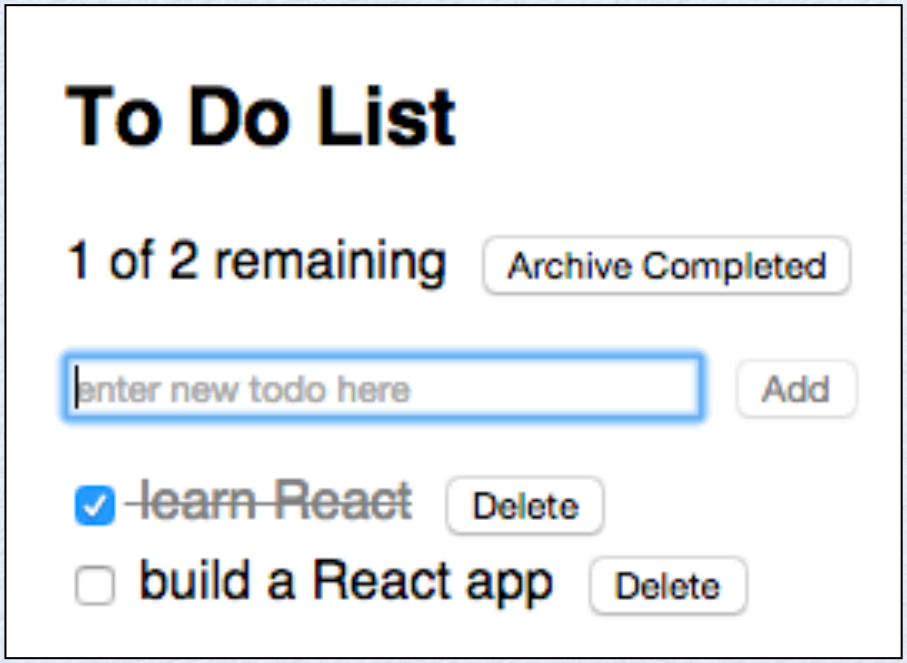
```
body {
  font-family: sans-serif;
  padding-left: 10px;
}

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:3000`

# ... Todo List App ...

todo.js

```
import {bool, func, shape, string} from 'prop-types';
import React from 'react';

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

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

export default Todo;
```

# ... Todo List App ...

todo-list.js

```
import React, {Component} from 'react';
import Todo from './todo';
import './todo.css';

let lastId = 0;

const createTodo = (text, done = false) => ({id: ++lastId, text, done});

class TodoList extends Component {
  state = {
    todoText: '',
    todos: [
      createTodo('learn React', true),
      createTodo('build a React app')
    ]
  };

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

# ... Todo List App ...

todo-list.js

```
onAddTodo = () =>
  this.setState(state => ({
    todoText: '',
    todos: state.todos.concat(createTodo(state.todoText))
  }));

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

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

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

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

just deleting in this simple version

# ... Todo List App

todo-list.js

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

    return (
      <div>
        <h2>To Do List</h2>
        <div>
          {this.uncompletedCount} of {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={todoText}
            onChange={this.onTextChange}/>
          <button disabled={!todoText}
            onClick={this.onAddTodo}>Add</button>
        </form>
        <ul className="unstyled">{todoElements}</ul>
      </div>
    );
  }
}

export default TodoList;
```

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

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

# Component Life Cycle

- Three phases
- **Mount**
  - component initialized and inserted into DOM
  - includes initial render
- **Update**
  - component re-rendered to virtual DOM and actual DOM updated if needed
  - triggered by state or prop changes
- **Unmount**
  - component removed from DOM
- **Lifecycle methods**
  - invoked in these specific phases
  - but most components don't use them

# Lifecycle Methods ...

most components don't use any of these

- **`componentDidMount()`**
  - invoked immediately after initial render
  - can perform setup such as
    loading initial data from an Ajax service
    or subscribing to data sources
- **`componentDidUpdate(prevProps, prevState)`**
  - called after updates are flushed to DOM, but not after initial render
- **`componentWillUnmount()`**
  - called immediately before a component is removed from DOM
  - good place to perform teardown
    such as unsubscribing from data sources

# … Lifecycle Methods …

- **`shouldComponentUpdate`**`(nextProps, nextState)`
  - not called before initial render, but before others
  - use to **optimize performance** by avoiding unnecessary virtual DOM creation and diffing
  - return `true` to proceed with render; `false` otherwise
  - subsequent lifecycle methods will not be called if this returns `false`

# … Lifecycle Methods

- static `getDerivedStateFromProps(nextProps, prevState)`
  - "invoked after a component is instantiated as well as when it receives new props"
  - "return an object to update state, or null
    to indicate that the new props do not require any state updates"

- `getSnapshotBeforeUpdate(prevProps, prevState)`
  - "invoked right before the most recently rendered output is committed to e.g. the DOM"
  - "enables your component to capture current values
    (e.g. scroll position) before they are potential changed"
  - "any value returned by this lifecycle will be passed as a parameter
    to `componentDidUpdate` in the newly supported 3rd parameter"

# Client-side State

- Options for holding client-side data ("state") used by components
- 1) Every component holds its own state
  - not recommended; hard to manage
- 2) Only a few top-level components hold state
  - these pass data to sub-components via props
- 3) "Store" hold state (ex. Redux)
  - useful when multiple components need access to the same data
  - useful when changes need to be persisted and restored later, such as when components are unmounted and later mounted again
- 4) Context API
  - somewhat new

# Managing State

- **Redux** is the most popular approach
  - http://redux.js.org/
  - supported by many libraries: react-redux, redux-logic, redux-saga, redux-thunk, ...
- Has many benefits, but also adds complexity and libraries on top of it add more
  - action objects
  - action type constants
  - action creator functions
  - dispatching actions
  - reducers
  - creating the store
  - providers that wrap the top component
  - connected components that listen for store changes
  - sagas
  - thunks

# Redux Overview

- "Predictable state container for JavaScript apps", not just for use with React
- Name is a contraction of "reducers" and "Flux"
  - a variation on the Flux architecture that uses a single "store"
- Resources
  - main website - http://redux.js.org
  - free video series from Dan Abramov - https://egghead.io/series/getting-started-with-redux
  - Dan Abramov talk - "Live React: Hot Reloading with Time Travel" from react-europe 2015 https://www.youtube.com/watch?v=xsSnOQynTHs

# Redux Three Principles

- Represent entire app state in a **single** JS object (**store**)
  - single source of truth
  - typically a deeply nested object
  - can include data describing UI state (ex. current sort order, filtering, ...)
- Only change state by **dispatching an action** to the store
  - unidirectional data flow
  - never directly modified from view
  - avoids race conditions where multiple code paths are taking turns modifying parts of the state

Only components that are unique to the app should dispatch events because doing that requires knowledge of the app state management and makes them non-reusable.

- Use **reducers** to derive new state from old state
  - functions that take current state and an action and return new state
  - composable - reducer functions can call other reducer functions that handle specific parts of the state tree

# Redux Actions

- Objects that contain **`type`** and **`payload`** properties
  - **`type`** values are typically constants with string values
  - **`payload`** can be an any kind of value
- Sent to store with **`store.dispatch(action)`**
  - see react-redux **`connect`** function for another way to dispatch actions
- For a Todo app, examples of actions include
  - add todo, mark todo completed, delete todo, archive completed todos, filter todos
- Can record a session by saving all actions
- Can replay a session by replaying saved actions

# Redux Reducers ...

- Pure functions that
  take current state and an action,
  and return new state
  - do not modify arguments
  - do not call functions that have side effects (ex. Ajax and route changes)
  - can use any of the immutability options discuss earlier
    to simplify creating a new state object from the current one
- Name comes from `Array` `reduce` method
  that takes a function with the same signature
  - `(accumulator, value) => accumulator`
- Easy to write tests for reducer functions
  - since they are pure

**Ajax calls** should be made by event handling functions or functions called by those. Results can be passed to Redux via actions so store can be updated.

# … Redux Reducers

- Reducers often contain a **`switch`** statement that switches on **`action.type`** and returns the current state object if they don't handle the passed action type
- By convention, if current state is **`undefined`** then return initial state

Example

```
function reducer(state = initialState, action) {
  const {todos} = state;
  const {type, payload} = action;
  switch (type) {
    case ADD_TODO:
      // payload is todo object to add
      return {
        ...state,
        todos: todos.concat(payload)
      };
    case DELETE_TODO:
      // payload is id of todo to delete
      return {
        ...state,
        todos: todos.filter(t => t.id !== payload)
      };
    ...
    default:
      return state;
  }
}
```

constants defined elsewhere

# Redux State

- Can include
  - locally created data
  - remotely created data such as from REST responses
  - UI state such as current route, selected tabs, expanded accordions, pagination details
- Advice on shape of state tree
  - keep UI state in a different part of tree from other data
  - store other data in a normalized fashion (like a relational database)
    - separate collections of objects by type rather than nesting (ex. employees, teams, projects)
    - key by ids to support referencing between objects

# Redux Typical Steps

- Define state shape
- For each action
  - define an action type constant
  - determine the type of the payload
  - possibly write an action creator function
  - add a `switch case` to some reducer function
  - dispatch the action from somewhere,
    possibly in an event handling function
    - can create the action object manually
      or call an action creator function

# Using Redux with React

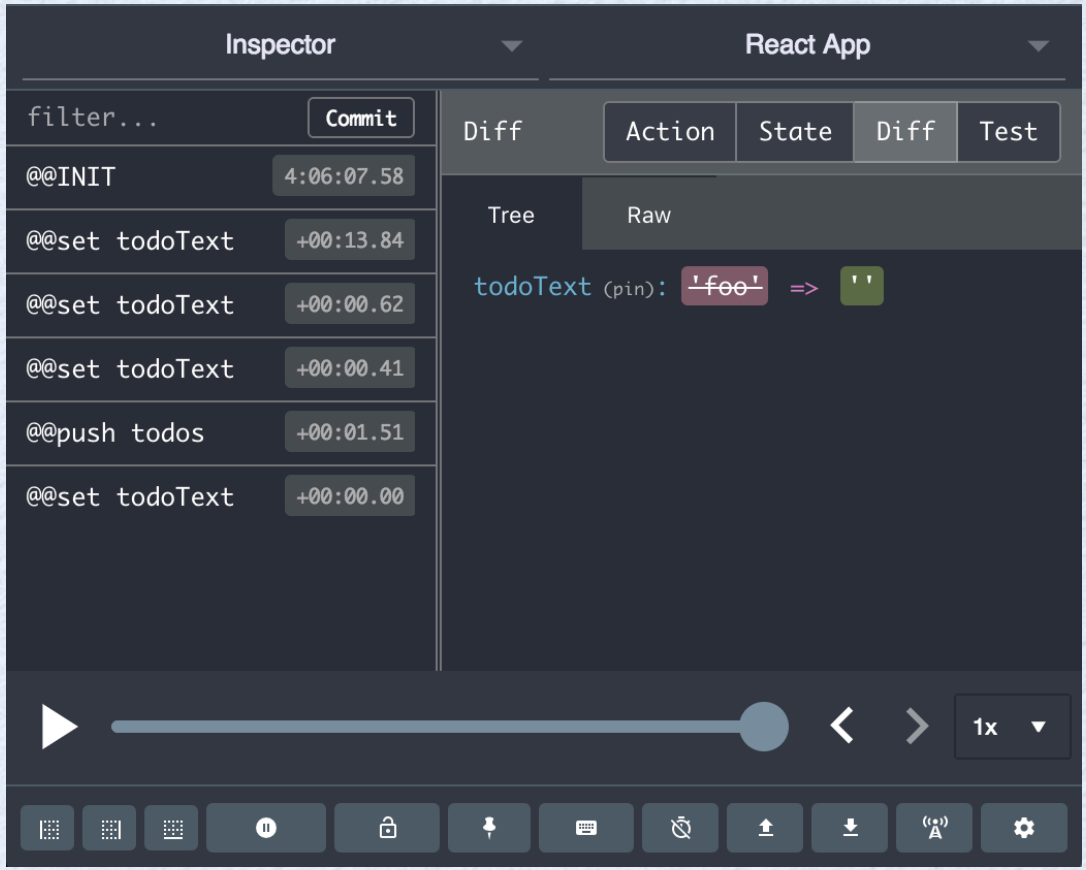
- There are many libraries for using Redux with React
- Most popular is react-redux at https://github.com/reduxjs/react-redux
- Steps to use
  - write top-level reducer function
    - can also have sub-reducers
  - create store
    - `const store = createStore(reducerFn);`
  - wrap top component in a **`Provider`** and render that
    - `<Provider store={store}> ... </Provider>`
  - in components
    - use `connect` function to create a higher-order component (HOC) that has access to the state and the `dispatch` function and export this instead of the raw component
    - any parts of the state can be made available to the component as props using the `mapStateToProps` function that is passed to `connect`
    - functions that dispatch actions can be made available to the component as props using the `mapDispatchToProps` function that is passed to `connect`

When `mapDispatchToProps` is not used, the `dispatch` function is made available to the component as a prop and can be called from event handling functions.

# Redux Devtools

https://github.com/reduxjs/redux-devtools

- Browser extension for Chrome and Firefox
- Can view
  - all actions that have been dispatched
  - payload of a given action
  - state after a given action
  - state diff created by a given action
- Can perform time-travel debugging
  - reset UI to any previous action
  - update UI to match its condition at the previous or next action



# redux-easy

https://www.npmjs.com/package/redux-easy

- Considerably easier than using Redux and react-redux directly!
  - but those are used under the covers
- Steps to use
  - define initial state
  - call `reduxSetup`, passing it the top component and the initial state
  - call `watch` to create higher-order components that are passed props for all state changes it cares about
  - optionally call `addReducer` to associate an action name with the function that handles it
    - only needed for complex actions
  - call `dispatch` functions to modify state
    - `dispatch`, `dispatchSet`, `dispatchTransform`, `dispatchDelete`, `dispatchPush`, `dispatchMap`, `dispatchFilter`
  - use provided components for basic form elements that are tied to state properties
    - `Input`, `TextArea`, `Select`, `RadioButtons`, `Checkboxes`

**Additional Benefits:**
- configures use of Redux Devtools
- saves all state changes in localStorage
- retrieves it on refresh

# Todo App ...

This version uses **redux-easy** to manage state instead of component state.

react-examples/todo-redux-easy

index.js

```
import React from 'react';
import {reduxSetup} from 'redux-easy';
import TodoList, {createTodo} from './todo-list';

const initialState = {
  todos: [
    createTodo('learn React', true),
    createTodo('build a React app')
  ],
  todoText: ''
};

reduxSetup({component: <TodoList />, initialState});
```

# ... Todo App ...

These can be plain functions when redux-easy is used.

todo.js

```
import {bool, shape, string} from 'prop-types';
import React from 'react';
import {dispatchFilter, dispatchMap} from 'redux-easy';

function onDeleteTodo(todoId) {
  dispatchFilter('todos', t => t.id !== todoId);
}

function onToggleDone(todo) {
  const id = todo.id;
  dispatchMap(
    'todos',
    t => (t.id === id ? {...t, done: !t.done} : t)
  );
}
```

# ... Todo App ...

todo.js

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

Todo.propTypes = {
  todo: shape({
    done: bool.isRequired,
    text: string.isRequired
  }).isRequired
};

export default Todo;
```

# .. Todo App...

todo-list.js

```
import {arrayOf, string} from 'prop-types';
import React, {Component} from 'react';
import {
  dispatchFilter, dispatchPush, dispatchSet, Input, watch
} from 'redux-easy';
import Todo from './todo';
import './todo.css';

let lastId = 0;

export const createTodo = (text, done = false) =>
  ({id: ++lastId, text, done});

const onArchiveCompleted = () => dispatchFilter('todos', t => !t.done);

class TodoList extends Component {
  static propTypes = {
    todos: arrayOf(Todo.propTypes.todo).isRequired,
    todoText: string.isRequired
  };

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

  onAddTodo = () => {
    dispatchPush('todos', createTodo(this.props.todoText));
    dispatchSet('todoText', '');
  };
```

just deleting in this simple version

# ... Todo App

todo-list.js

```
 render() {
    const {todos, todoText} = this.props;
    const todoElements = todos.map(todo => <Todo key={todo.id} todo={todo} />);
    return (
      <div>
        <h2>To Do List</h2>
        <div>
          {this.uncompletedCount} of {todos.length} remaining
          <button onClick={this.onArchiveCompleted}>Archive Completed</button>
        </div>
        <br />
        <form>
          <Input
            type="text"
            size="30"
            autoFocus
            path="todoText"
            placeholder="enter new todo here"
          />
          <button disabled={!todoText} onClick={this.onAddTodo}>
            Add
          </button>
        </form>
        <ul className="unstyled">{todoElements}</ul>
      </div>
    );
  }
}

export default watch(TodoList, {todos: '', todoText: ''});
```

# Context API

- Provider
  - a component that manages its own state
  - provide an object ("value") that exposes data and methods to update the data
- Consumer
  - uses one or more Providers to gain access to a object they provide
  - can use and render data in these objects
  - can call methods on these objects to update the data
- How many Providers?
  - can use one that provides and manages all app data (like a single Redux store)
  - can use multiple that each provide and manage specific subsets of the app data
- Requires using JSX that doesn't produce DOM

# Provider Example

```
import React, {Component} from 'react';

// Context is an object with Provider and Consumer properties.
export const Context = React.createContext();

class CounterProvider extends Component {
  state = {counter: 0};
  // Defining methods outside render method
  // so they aren't recreated on every render.
  methods = {
    increment: (event, delta = 1) =>
      this.setState(state => ({counter: state.counter + delta}))
  };

  render() {
    const value = {...this.state, ...this.methods};
    return (
      <Context.Provider value={value}>
        {this.props.children}
      </Context.Provider>
    );
  }
}

export default CounterProvider;
```

renders children of this component here; see the Counter element on slide 63

this exports two things, the Context and the custom Provider

# Consumer Example

```
import React, {Component} from 'react';
import {Context} from './counter-provider';
import {string} from 'prop-types';

class Counter extends Component {
  static propTypes = {
    label: string
  };
  static defaultProps = {
    label: 'Counter'
  };
  render() {
    return (
      <Context.Consumer>
        {context => (
          <div>
            <div>
              {this.props.label}: {context.counter}
            </div>
            <button onClick={context.increment}>+1</button>
            <button onClick={e => context.increment(e, 3)}>
              +3
            </button>
          </div>
        )}
      </Context.Consumer>
    );
  }
}

export default Counter;
```

a component that uses a Provider

passing a "function as child" which will be called with the value provided by the Provider

# App Setup Example

```
import React, {Component} from 'react';
import Counter from './counter';
import CounterProvider from './counter-provider';

class App extends Component {
  render() {
    // Multiple providers can be nested and
    // each can manage a separate part of the app state.
    return (
      <div className="App">
        <CounterProvider>
          <Counter label="My Label" />
        </CounterProvider>
     </div>
    );
  }
}

export default App;
```

This code isn't too bad when there is only one Provider, but it gets messy when there is more than one. In that case they need to be nested.

# REST Calls

- create-react-app supports the Fetch API out of the box
  - standard described at https://fetch.spec.whatwg.org/
- Example
  - suppose there was a REST service to retrieve all the todos, perhaps from a database

```
async function getTodos() {
  const url = '...some url...';
  const res = await fetch(url);
  return res.json();
}

class TodoList extends Component {
  ...
  async componentDidMount() {
    try {
      const todos = await getTodos();
      dispatchSet('todos', todos);
    } catch (e) {
      console.error(e);
    }
  }
  ...
}
```

call `res.text()` for text-based, non-JSON responses

can pass object with properties that specify the method (defaults to `'GET'`), body, headers, and more

lifecycle method

# Routing

- Maps URLs to components to render ... and more
- **react-router** is a popular choice
  - supports nested views
- For simple needs **react-hash-route** is easier
  - set example setup at https://www.npmjs.com/package/react-hash-route
  - **much simpler** to learn and use than react-router,
    yet it handles common routing needs
  - routing is a kind of configuration that is different from UI markup;
    **using JSX for this feels wrong**, so this library doesn't do that
  - makes it very **easy to change the route** inside a component method;
    just requires a call to the `route` function, passing it a hash name
  - nothing extra is required to support
    **remembering the route if the user refreshes** the browser;
    getting this to work with react-router seems tedious

# Jest

https://facebook.github.io/jest/

- Test framework from Facebook
- "Built on top of **Jasmine**"
- "Automatically finds tests to execute in your repo"
  - by default, all `.test.js` and `.spec.js` files in project and all `.js` files in and under `__tests__` directory
- "Runs tests with a fake DOM implementation (via **jsdom**) so that tests can run on the command line"
- **Watches** source and test files and automatically **reruns** tests when they change
  - can run all tests or only the ones that failed in the last run
- Can test **React components**
- Support **snapshot tests**
- Default test framework of apps created with **create-react-app**
  - `npm test` or `npm t` runs tests

# Jest API Highlights

- `describe(name, fn)`
  - describes a "test suite"
- `describe.only(name, fn)`
  - alias `fdescribe`
- `describe.skip(name, fn)`
  - alias `xdescribe`
- `beforeAll(fn)`
  - run once before all tests in suite begin
- `beforeEach(fn)`
  - run before each test in suite begins
- `afterEach(fn)`
  - run after each test in suite begins
- `afterAll(fn)`
  - run once after all tests in suite finish

`test` calls are not required to be inside a `describe`

- `test(name, fn)`
  - alias `it`
- `test.only(name, fn)`
  - alias `fit`
- `test.skip(name, fn)`
  - alias `xit`

- `expect(value)`
  - chain a matcher call onto this

matchers are described next

# Jest Matchers ...

See https://facebook.github.io/jest/docs/api.html#writing-assertions-with-expect

- `.not` - can be prepended to any matcher
- `.toBe(value)` - uses ===
- `.toEqual(value)`
  - deep object comparison
- `.toBeTruthy()`
- `.toBeFalsy()`
- `.toBeDefined()`
- `.toBeUndefined()`
- `.toBeNull()`
- `.toMatch(regexp)`
- `.toMatchObject(object)`
  - all properties in *object* match those in receiver
- `.toBeCloseTo(number, digits)`
- `.toBeGreaterThan(number)`
- `.toBeGreaterThanOrEqual(number)`
- `.toBeLessThan(number)`
- `.toBeLessThanOrEqual(number)`
- `.toBeInstanceOf(Class)`
- `.toContain(item)`
- `.toContainEqual(item)`
  - deep object comparison
- `.toHaveLength(number)`

for arrays

# ... Jest Matchers

- `.toHaveBeenCalled()`
  - alias is `.toBeCalled`
- `.toHaveBeenCalledTimes(number)`
- `.toHaveBeenCalledWith(arg1, arg2, ...)`
  - alias is `.toBeCalledWith`
- `.toHaveBeenLastCalledWith(arg1, arg2, ...)`
  - alias is `.lastCalledWith`

- `.toThrow()`
- `.toThrowError(error)`

- `.toMatchSnapshot()`
- `.toThrowErrorMatchingSnapshot()`

# Promise Matchers

- Add `.resolves` or `.rejects` before other matchers that test a value
- Example
  - assume that `makeRestCall()` returns a `Promise`
  - `expect(makeRestCall()).resolves.toBe(value);`
  - `expect(makeRestCall()).rejects.toMatch(substring);`
- Alternatively
  - mark test function as `async`
  - put test code inside a `try/catch`
  - use `await` with function calls that return a `Promise`

# Async Tests

- Function passed to tests of asynchronous functions should have a `done` parameter
- When all asynchronous actions have completed, call `done()`
- Can fail a test by calling `done.fail(error)`
- Example

```
try {
  expect(actual).toBe(expected);
  done();
} catch (e) {
  done.fail(e);
}
```

but throwing already causes a test to fail, so don't need to do this in a `catch`

# More Jest Details

- Errors in tests
  - if code inside a test throws, the test fails
  - can be used as an alternative to explicit assertions
- Testing functions that return promises
  - if a test returns a promise,
    it will wait for the promise to resolve or reject
    and fail if it rejects
- Test data
  - consider using faker.js npm package to generate
  - https://github.com/Marak/Faker.js

# react-testing-library ...

https://github.com/kentcdodds/react-testing-library

- Adds features to Jest
- Alternative to **Enzyme** for testing React components
  - some prefer this because the test code is closer to what other developers would write in actual usage
- Builds on features of **dom-testing-library**
  - also from Kent C. Dodds
- Makes it easy to find elements in many ways
- To install
  - `npm install -D react-testing-library jest-dom`
  - `jest-dom` adds DOM-related assertions for Jest

# … react-testing-library …

https://facebook.github.io/jest/

- To render the component to be tested
  - `const renderResult = render(componentJSX);`
- Object returned by `render` contains
  - properties
    - `container` - container element created to hold component (a `div` by default)
    - `baseElement` - element to which container is appended (`document.body` by default)
  - functions
    - `debug()` - outputs an HTML description of the DOM produced
    - `rerender(jsx)` - to test updates to component props
    - `unmount()` - to test what happens when component is removed from page
    - (`get`|`query`)(`All`)?`By`{`kind`}
      where `kind` is nothing, `AltText`, `LabelText`, `PlaceholderText`, `Text`, `TestId`, `Title`, or `Value`

`debug()` is equivalent to
`console.log(prettyDOM(container));`
where `prettyDOM` comes from dom-testing-library

`AltText` is for `img` elements.
`TestId` finds elements by `data-testid` attribute value.

`get` functions throw if target is not found.
`query` functions return null if target is not found

Functions with "`All`" return an array of matches.
otherwise only first match is returned.

Finding elements by their `data-testid` attribute value is better than relying on element nesting or CSS class names since those can change and cause tests to break.

# … react-testing-library …

- Custom Jest Matchers
  - to get these, `import 'jest-dom/extend-expect';`
  - adds methods to object returned by `expect` function
  - `toBeInTheDOM()`
  - `toHaveAttribute(name [, value])`
  - `toHaveClass(name)`
  - `toHaveStyle(cssPropertiesString)`
    - an example CSS properties string is
      `'color: red; display: inline-block'`
  - `toHaveTextContent(text)`
  - `toBeVisible()`
    - **visible** means
      `display` is not `none`,
      `visibility` is not `hidden` or `collapse`,
      `opacity` is not `0`,
      and all ancestors are visible
  - can add `.not` before each of these

# ... react-testing-library

- To fire events
  - `fireEvent(node, eventObject)`
  - `fireEvent.{kind}(node, eventProperties)`
    - *kind* includes `change`, `click`, `dblClick`, `keyDown`, and many more
    - for list of supported events, see `eventMap` at https://github.com/kentcdodds/dom-testing-library/blob/master/src/events.js
- To find an element only within another
  - import the within function from react-testing-library
  - `const element = within(parentElement).get...(...);`
- To wait for an element be rendered before running other code
  - `waitForElement(callback)`
  - useful when an element to be tested is not rendered until some asynchronous action such as a REST call completes

# Example Test

demo.js

```
import React, {Component} from 'react';

class Demo extends Component {
  state = {count: 0};

  onClick = () => this.setState(
    state => ({
      ...state,
      count: state.count + 1
    })
  );

  render() {
    return (
      <div className="demo">
        <button onClick={this.onClick}>
          Press Me
        </button>
        <br />
        <label>Count:</label>
        <span data-testid="count">
         {this.state.count}
        </span>
      </div>
    );
  }
}

export default Demo;
```

demo.test.js

```
import 'jest-dom/extend-expect';
import React from 'react';
import {cleanup, fireEvent, render}
from 'react-testing-library';
import Demo from './demo';

// Automatically unmount and
// clean up DOM after each test.
afterEach(cleanup);

test('renders without crashing', () => {
  render(<Demo />);
});

test('button works', () => {
  const {getByTestId, getByText} =
    render(<Demo />);
  const countSpan = getByTestId('count');
  expect(countSpan).toHaveTextContent('0');

  const button = getByText('Press Me');
  fireEvent.click(button);
  expect(countSpan).toHaveTextContent('1');
});
```

# Biggest Benefits of React

- Emphasizes using JavaScript
  - rather than using custom template syntax to build component views
- Components are easy to define
  - can be a single function or a class that extends `Component` and has a `render` method
- 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
- Component tests are easy to write
- Same approach can be used for many targets
  - DOM, Canvas, SVG, Android, iOS, ...
- Widely used and well-supported
  - easy to find developers, libraries, example code, and training material

# Wrap Up

- Client-side recommendations
  - React with create-react-app
  - Sass CSS preprocessor
  - redux-easy
- Recommendations for both sides
  - ESLint
  - Prettier

# Hands On Exercise #1

- Let's add a feature to the "Tour of Heroes" web UI!
- Enable filtering of heroes in "Heroes" view

solution is in "filter" branch

  - edit `client/src/types.js`
    - add "`filter: string,`" to `StateType`
  - edit `client/src/initial-state`
    - add "`filter: '',`" to `initialState`
  - edit `render` method in `client/src/hero-list.js`
    - add `filter` to destructuring of `this.props`

```
const {filter, heroes} = this.props;
```

    - change assignment to `heroList` to use `let` instead of `const`
    - add line to filter `heroList` if a `filter` has been entered

```
let heroList = heroMapToList(heroes);
if (filter) heroList = heroList.filter(
  hero => hero.name.includes(filter));
```

    - add a `label` and `Input` (from redux-easy) to the `render` output immediately before the list of heroes

```
<div>
  <label>Filter</label>
  <Input path="filter" />
</div>
```

# Hands On Exercise #2

- Create a component that displays statistics about the heroes

solution is in "statistics" branch

  - create a directory named **`hero-statistics`** under **`src`** to hold the files for this component
  - create the files **`hero-statistics.js`** and **`hero-statistics.scss`** in this directory
  - name the component **`HeroStatistics`**
  - display the following
    - number of heroes
    - name of the hero with the shortest name
    - name of the hero with the longest name

**# of heroes:** 10
**Shortest Name:** Narco
**Longest Name:** Celeritas

  - display this new component after the buttons in the **`render`** method of **`App.js`**
  - see tips on following slides

# hero-statistics.js …

```
// @flow

import React, {Component} from 'react';
import {watch} from 'redux-easy';

import {type HeroMapType, heroMapToList} from '../types';

import './hero-statistics.css';

type PropsType = {heroes: HeroMapType};

function getLongestName(nameList) {
  if (nameList.length === 0) return '';
  return nameList.reduce(
    (longest, name) => (name.length > longest.length ? name : longest),
    ''
  );
}

function getShortestName(nameList) {
  if (nameList.length === 0) return '';
  const [firstName] = nameList;
  return nameList.reduce(
    (shortest, name) => (name.length < shortest.length ? name : shortest),
    firstName
  );
}
```

# … hero-statistics.js

can implement with a function component

```
const HeroStatistics = (props: PropsType) => {
  const nameList = heroMapToList(props.heroes).map(hero => hero.name);
  return (
    <div className="hero-statistics">
      {/* render a label and the number of heroes */}
      {/* render a label and the shortest name */}
      {/* render a label and the longest name */}
    </div>
  );
};

export default watch(HeroStatistics, {heroes: ''});
```

# hero-statistics.scss

```
.hero-statistics {
  border: solid red 2px;
  border-radius: 4px;
  display: inline-block;
  margin-top: 10px;
  padding: 4px;

  label {
    display: inline-block;
    font-weight: bold;
    text-align: right;
    width: 120px;
  }
}
```