

React

R. Mark Volkmann

Object Computing, Inc.

<http://objectcomputing.com>

Email: mark@objectcompuing.com

Twitter: @mark_volkman

GitHub: mvolkmann



OCI

WE ARE
SOFTWARE
ENGINEERS.

Outline

- React Overview
- create-react-app Tool
- Components
- JSX
- Props
- State
- Events
- Sample App
- Managing State
- redux-easy Library
- Wrap-up

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 and components a function of their “props”
 - like “pure functions” in functional programming
- Supports Chrome, Edge, Firefox, IE9+, and Safari

... 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. Canvas, SVG, Android, 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>

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.”

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!**
 - let Facebook maintain and improve the build process

or skip install and run
`npx create-react-app app-name`



Welcome to React

To get started, edit `src/App.js` and save to reload.

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**
- Produces small production deploys
- Lets Facebook maintain the build process
 - future benefits from future improvements



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 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

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

```
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;
```

"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`

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* }
- 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. **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

not statements!
ex. ternary instead of **if**

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

```
import React from 'react';  
  
class Greeting extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}!</h1>;  
  }  
}  
  
export default Greeting;
```

src/greeting.js

```
import React from 'react';  
  
export default ({name}) =>  
  <h1>Hello, {name}!</h1>;
```

props object is passed
and destructured

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

src/App.js

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

inside render method

Hello, Mark!

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
- 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)

objectOf is useful when an object is used as a map of key/value pairs where the values all have a common type

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
- To add/modify state properties, call **this.setState**

```
state = {age: 0, name: ''};
```

- **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 {age, name} = this.state;
```

- Never directly modify **this.state**

can catch in ESLint with
react/no-direct-mutation-state rule

- can cause subtle bugs
- see <https://reactjs.org/docs/react-component.html#setState>

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 to 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={e => this.handleClick(e)}`
- Passed a React-specific event object
 - `target` property refers to React component where event occurred

hipsters might do this:

```
onClick={this.handleClick.bind(this)}
```

State/Event Example ...

react-examples/event-demo

Name:
Hola, World!

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

defined on next slide

... State/Event Example

```
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;
```

src/greeting.js

optional prop validation

The diagram illustrates the state and prop flow in the `Greeting` component. It shows three main connections: 1. An arrow from the comment `// initial state` to the `state` property of the class. 2. An arrow from the `setName` method to the `this.setState` call within it. 3. An arrow from `this.state.name` in the `value` prop of the `<input>` element to the `state` property, indicating that the input's value is derived from the current state.

Todo App ...

react-examples/todo

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

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

index.js

modified from what
create-react-app generated

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

todo.css

```
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 Do List

1 of 2 remaining

Archive Completed

enter new todo here

Add

☒ -learn React

Delete

☐ build a React app

Delete

To run:
npm start
browse localhost:3000

... Todo App ...

```
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.js

... Todo App ...

```
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.js

... Todo App ...

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

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

just deleting in this simple version

```
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};
  });
```

todo-list.js

Many of these methods would be easier if todos were held in a map keyed by todo id instead of an array.

... Todo 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>
  );
}
```

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.

```
export default TodoList;
```


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 type constants
 - action objects
 - 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-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 sessionStorage
- retrieves it on refresh

Todo App ...

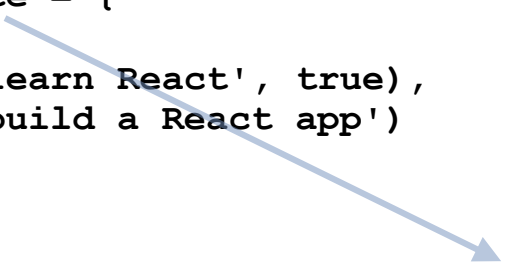
This version uses **redux-easy** to manage state instead of component state.

react-examples/todo-redux-easy

```
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});
```

index.js



... Todo App ...

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

```
import {bool, shape, string} from 'prop-types'; todo.js
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 ...

```
// A props object is passed to this function and destructured.
const Todo = ({todo}) => (                                     todo.js
  <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 ...

```
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', '');
  };
}
```

todo-list.js

just deleting in this simple version

see use of **watch** at bottom of next slide that makes these available to this component as props

... 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: ''});
```


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

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

```
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);  
    }  
  }  
  ...  
}
```

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

lifecycle method

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

Big Questions

- Is it easier to learn and use React than other options?
- 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 ask me questions about React or anything in the JavaScript world

- **Contact me**

Mark Volkmann, Object Computing, Inc.

Email: mark@objectcomputing.com

Twitter: [@mark_volkmann](https://twitter.com/mark_volkmann)

GitHub: [mvolkmann](https://github.com/mvolkmann)

Website: <https://mvolkmann.github.io/>