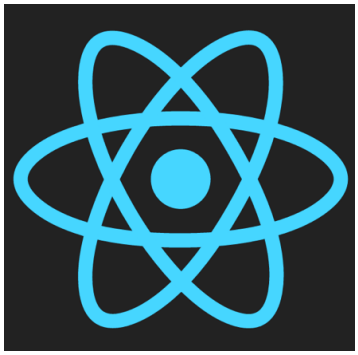


Introduction to React

version 1.7, January 2019

Introduction

Objectives

- Who am I?
- Who are you?
- Daily Schedule
- Course Schedule and Syllabus

About Me

Introductions

- What's your name?
- What do you do?
- JavaScript level (beginner, intermediate, advanced)?
- What do you want to know at the end of this course?
- Favorite food?

Daily Schedule

- 08:30 - 10:30
- 15 minute break
- 10:45 - 12:00
- 1 hour lunch break
- 1:00 - 2:00
- 15 minute break
- 2:15 - 3:15
- 15 minute break
- 3:30 - 4:30

The Big Picture

- Topics Covered in this Course
  - React.js
  - Modern JavaScript (ECMAScript 2017 and beyond)
  - Testing with Jest and Enzyme
  - AJAX with React
  - Flux / Redux
  - Server-side React Rendering

Course Homepage

- http://watzthisco.github.io/intro-to-react
- This is the homepage for the course, which contains the final application created in the labs.
- If you discover any bugs or typos, please report them:
  - https://github.com/watzthisco/intro-to-react/issues

What is React.js?

- A library for building user interfaces
- Created by Facebook
  - Open sourced in 2014
- Abstracts the Document Object Model (Virtual DOM)
- Implements one-way data flow
- Component-based
- Can be rendered on the server
- Plays well with other libraries / frameworks

What is React NOT?

- React is not a framework.
- React doesn't have AJAX capabilities.
- React has no data layer.

When can you use React?

- Complex single-page applications (SPAs) can be built entirely using React.
- React can be used as a substitution for views in a traditional MV* framework.
- React can generate static HTML on the server.
- React can be used to create native mobile apps.

Who Uses React?

- AddThis
- Angie's List
- Airbnb
- Atlassian
- BBC
- Codecademy
- Coursera
- Dropbox
- Expedia
- Facebook
- IMDb
- Imgur
- Instagram
- Intuit
- Liberty Mutual
- Lyft
- Netflix
- NFL
- OkCupid
- Paypal
- Reddit
- Salesforce
- Squarespace
- Tesla Motors
- The New York Times
- Trulia
- Trunk Club
- Twitter
- Uber
- Visa
- WhatsApp
- Wired
- Wix
- Wolfram Alpha
- Wells Fargo
- WordPress
- Yahoo
- Zendesk

and many more!

React Quick Start

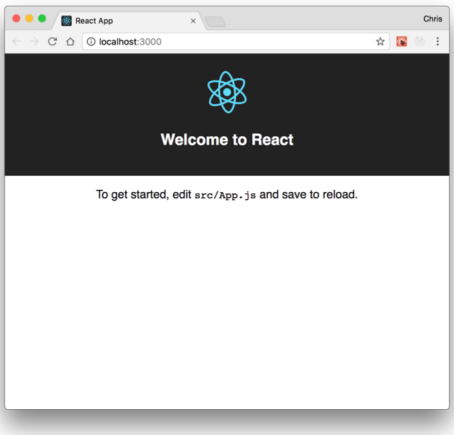
Objectives

- Install CreateReactApp
- Create a React App
- Test and Run a React App with CRA

What is Create React App

- Creates React App with no build configuration
- Simplifies setup for single-page apps and learning
- Sacrifices flexibility for simplicity
- Not a substitute for understanding the tools, but great when you just want to write React code quickly
- Other options offer more flexibility, less simplicity
  - nwb
  - Neutrino

Lab 01: Get Started with Create React App



Chapter 2:
Advanced JavaScript

Objectives

- Learn what's new
- Use arrow functions and block-scoped variables
- Create generator functions
- Use classes and modules

Variable Scoping with `const` and `let`

- `const` creates constants
  - "immutable variables"
  - Cannot be reassigned new content
  - The assigned content isn't immutable, however,
    - If you assign an object to a constant, the object can still be changed.
- `let` creates block-scoped variables
  - Main difference between `let` and `var` is that the scope of `var` is the entire enclosing function.
  - Redeclaring a variable with `let` raises a syntax error
  - No hoisting
    - Referencing a variable in the block before the declaration results in a ReferenceError.

let vs. var

var

```
var a = 5;
var b = 10;

if (a===5) {
  var a = 4;
  var b = 1;
}
console.log(a); // 4
console.log(b); // 1
```

let

```
let a = 5;
let b = 10;

if (a===5) {
  let a = 4;
  let b = 1;
}
console.log(a); // 5
console.log(b); // 10
```

Block-scoped Functions

ES5

```
(function () {
  var foo = function () {
  return 1;
  }
  console.log(foo()); // 1

  (function () {
    var foo = function() {
    return 2;
    }
    console.log(foo()); // 2
    })();

  console.log(foo()); // 1
})();
```

ES6

```
{
  function foo () { return 1; }
  console.log(foo()); // 1
  {
    function foo () {
    return 2;
    }
    console.log(foo()); // 2

  }
    console.log(foo()); // 1
}
```

Arrow Functions

- More expressive closure syntax
  - ES6

```
odds = evens.map (v => v+1);
```

  - ES5

```
odds = evens.map (function (v) { return v+1; });
```

Arrow Functions (cont.)

- More intuitive handling of current object context.
  - ES6

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v);
});
```

  - ES5

```
var self = this;
this.nums.forEach(function (v) {
  if (v % 5 === 0)
    self.fives.push(v);
});
```

Default Parameter Handling

- ES6

```
function myFunc (x, y = 0, z = 13) {
  return x + y + z;
}
```

- ES5

```
function f (x, y, z) {
    if (y === undefined)
        y = 0;
    if (z === undefined)
        z = 13;
    return x + y + z;
};
```

Rest Parameter

- Aggregation of remaining arguments into single parameter of variadic functions.

```
function myFunc (x, y, ...a) {
    return (x + y) * a.length;
}
console.log(myFunc(1, 2, "hello", true, 7));
```

- http://jsbin.com/pisupa/edit?js,console

Spread Operator

- Spreading of elements of an iterable collection (like an array or a string) into both literal elements and individual function parameters.

```
var params = [ "hello", true, 7 ];
var other = [ 1, 2, ...params ];
console.log(other); // [1, 2, "hello", true, 7]

console.log(MyFunc(1, 2, ...params));

var str = "foo";
var chars = [ ...str ]; // [ "f", "o", "o" ]
```

- http://jsbin.com/guxika/edit?js,console

Template Literals

- String Interpolation

```
var customer = { name: "Penny" }
var order = { price: 4, product: "parts", quantity: 6 }
message = `Hi, ${customer.name}. Thank you for your order
of ${order.quantity} ${order.product} at ${order.price}.`;
```

- http://jsbin.com/pusako/edit?js,console

Template Literals (cont.)

- Custom Interpolation
- Expression interpolation for arbitrary methods

```
get`http://example.com/cart?order=${orderId}`;
```

Template Literals (cont.)

- Raw String Access

Allows you to access the raw template string content (without interpreting backslashes)

```
function tag(strings, ...values) {
  console.log(strings.raw[0]);
  // "string text line 1 \\n string text line 2"
}
tag`string text line 1 \n string text line 2`;
```

- http://jsbin.com/donibif/edit?js,console

Enhanced Object Properties

- Property Shorthand
  - Shorter syntax for properties with the same name and value
- ES5

```
obj = { x: x, y: y};
```

- ES6

```
obj = { x,y };
```

Enhanced Object Properties

- Computed names in object property definitions

```
let obj = {
  customer: "Nigel",
  [ "order" + getOrderNum() ]: 10
};
```

- http://jsbin.com/wejuqe/edit?js,console

Method notation in object property definitions

ES6

```
obj = {
    foo (a,b) {
    },
    bar (x,y) {
    }
};
```

ES5

```
obj = {
    foo: function(a, b) {
    },
    bar: function(x, y) {
    }
};
```

Array Matching

- Intuitive and flexible destructuring of Arrays into individual variables during assignment

```
var list = [ 1, 2, 3 ];
var [ a, , b ] = list; // a = 1 , b = 3
[ b, a ] = [ a, b ];
```

- http://jsbin.com/yafage/edit?js,console

Object Matching

- Flexible destructuring of Objects into individual variables during assignment

```
var { a, b, c} = {a:1, b:2, c:3};
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- http://jsbin.com/kuvizu/edit?js,console

Symbol Primitive

- A **unique** and **immutable** data type.
- May be used as an identifier for object properties.
- Examples:
  - var sym1 = Symbol();
  - var sym2 = Symbol("foo");
  - var sym3 = Symbol("foo"); // Symbol("foo") !== Symbol("foo")
- Well-known Symbols
  - Built-in symbols, for example Symbol.iterator, which returns the default iterator for an object.

User-defined Iterators

- Customizable iteration behavior for objects
- In order to be iterable, an object must implement the iterator method -- the object, or one of the objects up its prototype chain) must have a property with a Symbol.iterator key.

```
var myIterable = {}
myIterable[Symbol.iterator] = function* () {
    yield 1;
    yield 2;
    yield 3;
};
[...myIterable] // [1, 2, 3]
```

For-Of Operator

- Convenient way to iterate over all values of an iterable object

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1
    return {
      next() {
        [ pre, cur] = [cur, pre + cur]
        return { done: false, value: cur }
        }
      }
    }
}
for (let n of fibonacci) {
    if (n > 1000)
      break;
    console.log(n);
}
```

- http://jsbin.com/nururoz/edit?js,console

Creating and Consuming Generator Functions

- A generator is a special type of function that works as a factory for iterators.

```
function* idMaker(){
  var index = 0;
  while(true)
    yield index++;
}
var gen = idMaker();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
```

- http://jsbin.com/pozite/edit?js,console

Class Definition

- ES6 introduces more OOP-style classes
- Can be created with Class declaration or Class expression

Class Declaration

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

Class Expressions

- Can be unnamed

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

- Or named

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

Class Inheritance

```
class Rectangle extends Shape {
    constructor (id, x, y, width, height) {
      super (id, x, y);
      this.width = width;
      this.height = height;
    }
}
class Circle extends Shape {
    constructor (id, x, y, radius) {
      super (id, x, y);
      this.radius = radius;
    }
}
```

Understanding `this`

- Allows functions to be reused with different context.
- Which object should be focal when invoking a function.

4 Rules of `this`

- Implicit Binding
- Explicit Binding
- New Binding
- Window Binding

What is `this`?

- When was function invoked?
- We don't know what `this` is until the function is invoked.

Implicit Binding

- `this` refers to the object to the left of the dot.

```
var author = {
  name: 'Chris',
  homeTown: 'Detroit',
  logName: function() {
    console.log(this.name);
  }
});

author.logName();
```

Explicit Binding

- .call, .apply, .bind

```
var logName = function() {
  console.log(this.name);
}

var author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName.call(author);
```

Explicit Binding with .call

- Calls a function with a given `this` value and the arguments given individually.

```
var logName = function(lang1) {
  console.log(this.fname + this.lname + lang1);
};
var author = {
  fname: "Chris",
  lname: "Minnick"
};
var language = ["JavaScript","HTML","CSS"];
logName.call(author,language[0]);
```

Explicit binding with .apply

- Calls a function with a given `this` value and the arguments given as an array

```
logName = function(food1,food2,food3) {
  console.log(this.fname + this.lname);
  console.log(food1, food2,
    food3);
};
var author = {
  fname: "Chris",
  lname: "Minnick"
};

var favoriteFoods= ['Tacos','Soup','Sushi'];

logName.apply(author, favoriteFoods);
```

Explicit Binding with .bind

- Works the same as .call, but returns `new` function rather than immediately invoking the function

```
logName = function(food) {
  console.log(this.fname +" " + this.lname +
        "\'s Favorite Food was " + food);
};
var person = {
  fname: "George",
  lname: "Washington"
};
var logMe = logName.bind(person);

logMe("Tacos");
```

- http://jsbin.com/xikuzog/edit?js,console

`new` Binding

- When a function is invoked with the `new` keyword, then this keyword inside the object is bound to the new object.

```
var City = function (lat,long,state,pop) {
  this.lat = lat;
  this.long = long;
  this.state = state;
  this.pop = pop;
};
var sacramento = new City(38.58,121.49,"CA",480000);
console.log (sacramento.state);
```

`window` Binding

- What happens when no object is specified or implied
- `this` defaults to the `window` object

```
var logName = function() {
  console.log(this.name);
}
var author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName(); //undefined(error in 'strict' mode)
window.author = "Harry";
logName(); // "Harry"
```

`Array.map()`

- **Array.map()**
  - Creates a new array with the results of calling a provided function on every element in this array.
- Syntax
  - var new_array = arr.map(callback)
- Parameters passed to the callback
  - `currentValue`
    - The current element being processed
  - `index`
    - The index (number) of the current element
  - `array`
    - The array map was called upon

PROMISES

What Are Promises?

- An abstraction for asynchronous programming
- Alternative to callbacks
- A promise represents the result of an async operation
- Is in one of three states
  - pending – the initial state of a promise
  - fulfilled – represents a successful operation
  - rejected – represents a failed operation

Promises vs. Event Listeners

- Event listeners are useful for things that can happen multiple times to a single object.
- A promise can only succeed or fail once.
- If a promise has succeeded or failed, you can react to it at any time.
  - `readJSON(filename).then(success,failure);`

Why Use Promises?

- Chain them together to transform values or run additional async actions
- Cleaner code
  - Avoid problems associated with multiple callbacks
    - Callback Hell
    - Christmas Tree
    - Tower of Babel
    - Etc.

Demo: Callback vs. Promise

- Callback
  - ```
    fs.readFile('text.txt', function(err, file){
        if (err){
            //handle error
        } else {
            console.log(file.toString());
        }
    });
    ```
- Promise
  - ```
    readText('text.txt')
        .then(function(data) {
                console.log(data.toString());
            }, console.error
        );
    ```

Using Promises

```
const fs = require('fs');

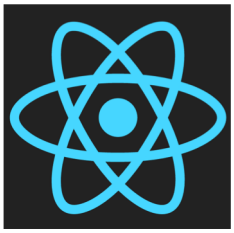
function readFileAsync (file, encoding) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, encoding, function (err, data) {
            if (err) return reject(err);
            resolve(data);
        })
    })
}

readFileAsync('myfile.txt')
    .then(console.log, console.error);
```

Introduction to React.js

Objectives

- Understand React
- Explore the Virtual DOM
- Explain one-way data binding
- Create React Components
- Manage state



What is React.js?

- A JavaScript library for building UIs
- Wraps an imperative API (DOM) with a declarative one
- Is comparable to Angular Directives
- Can be plugged into a framework's component technology
- Doesn't have to be used with MVC

Imperative API vs. Declarative API

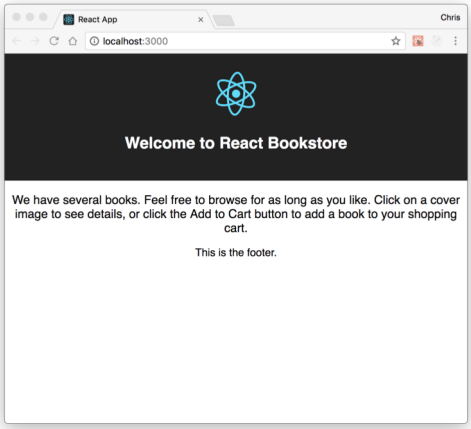
- Imperative
  - Focuses on the steps to complete a task
  - Example:
    - Walk to the stairs
    - Walk down stairs
    - Go to the kitchen
    - Open refrigerator
    - Take out salami, cheese, mustard
    - Put salami, cheese, mustard on bread
- Declarative
  - Focuses on what to do without saying how
    - Bring me a sandwich.

Imperative vs. Declarative Screen Updates

- Imperative
  - Change the greeting to: "Dear Mr. Smith,”
  - Change the beginning of the first paragraph to "We are pleased to”
  - Changing the closing to "Sincerely,”
- Declarative
  - Make it look like this:

```
Dear Mr. Smith,
We are pleased to inform you that you have been
selected!
Sincerely,
The Management
```

Lab 02: Your First Component

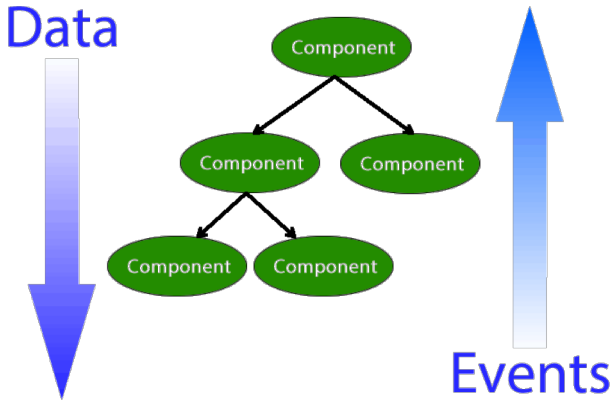


Key Points

- One-way data flow
  - A component can't modify properties passed to it.
- Virtual DOM
  - Manages HTML DOM updates
- JSX
  - Easy XML template language.
- Not just for browser output
  - Architecture can apply to native apps, canvas

One-way Data Flow

- Each UI element represents one component
- All data flows from owner to child



Virtual DOM

1. Virtual DOM is updated (in memory) as the **state** of the data model changes.
2. React calculates the difference between the Virtual DOM and the real DOM.
3. React updates only what needs to be updated in the DOM.
4. Batches changes

Virtual DOM vs. HTML DOM

- Virtual DOM is a local and simplified copy of the HTML DOM
- (It's an abstraction of an abstraction)
- The goal of the Virtual DOM is to only re-render when the **state** changes.
  - This makes it more efficient than direct DOM manipulation.
- Developers can write code as if the entire tree is being re-rendered.
  - This makes it easier to understand.
- Behind the scenes, React/Virtual DOM works out the details and creates a patch for the HTML DOM, which causes the browser to re-render the changed part of the scene.

State Machines

- React thinks of UIs as being simple state machines.
- A state machine has:
  - An initial state or record of something stored someplace
  - A set of possible input events
  - A set of new states that may result from the input
  - A set of possible actions or output events that result from a new state

Understanding Components

- Created by extending `React.Component`
  - Pass in an object with a render method
  - Returns a single element OR an Array of elements (as of v16)
    - May contain nested elements, however.
  - You can have multiple instances of components
    - For example, you might have components called NavButton or Invitee
  - An element describes a component and tells React what you want to see on screen.
- Components should be reusable as well as composable.

`React.render()`

- `fn(d) = V`
- Give the function data and you get a View.
- Return value must
  - be a single element:
    - `return (<div><p><em></em></p></div>);` = correct
    - `return (<div></div><p></p>);` = error
  - or an array with unique key values
    - ```
      return [
            <li key="1">item 1</li>
            <li key="2">item 2</li>
        ];
      ```
  - or a string
    - `return "this is totally valid too";`
      ```

Lab 03: Create More Components

ReactDOM

- React package for working with the DOM
  - Generally only needed at the top level of your application.
- Methods
  - `findDOMNode`
  - `render`
  - `unmountComponentAtNode`
    - Removes a mounted component from the DOM and cleans up its event handlers and state.
  - react-dom/server
    - For rendering static markup on the server.
    - `ReactDOMServer.renderToString()`
    - `ReactDOMServer.renderToStaticMarkup()`

ReactDOM.findDOMNode

- `findDOMNode(component)`
- If the component has been mounted, returns the corresponding native browser DOM element

ReactDOM.unmountComponentAtNode

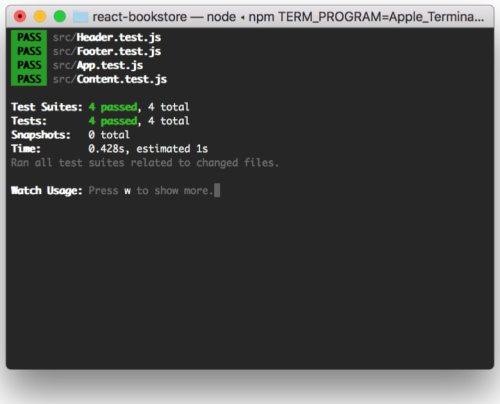
- Removes a mounted component from the DOM and cleans up its event handlers and state

```
ReactDOM.unmountComponentAtNode
    (document.getElementById('container'));
```

`ReactDOM.render`

- `ReactDOM.render(reactElement, domContainerNode)`
- Renders a `reactElement` into the DOM in the supplied container
- Replaces any existing DOM elements inside the container node when first called
- Later calls using DOM diffing algorithm for efficient updates

Lab 04: Testing React

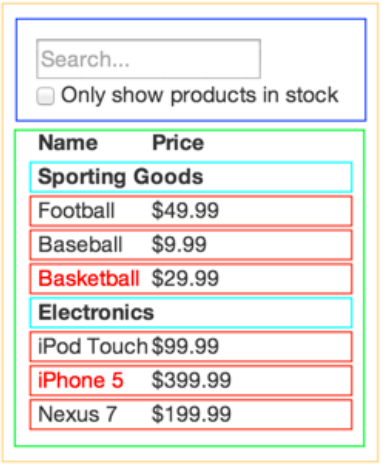


React Development Process

1. Break the UI into a component hierarchy
2. Build a static version in React using props
   - Props pass data from parent to child
3. Identify the minimal representation of UI state
4. Identify where your state should live
5. Add inverse data flow

Step 1 - Break up the UI

```
var PRODUCTS = [
  {category:
'Sporting Goods',
price: '$49.99',
stocked: true, name:
'Football'},
  {category:
'Sporting Goods',
price: '$9.99',
stocked: true, name:
'Baseball'}
];
```



Step 2 - Static Version

- Render the UI from the data model with no interactivity
- Goal is to have a library of reusable components that render the data model

```
class ProductCategoryRow extends React.Component({
  render () {
    return (
    <tr>
      <th colSpan="2">
        {this.props.category}
      </th>
    </tr>
    );
  }
});
```

What is JSX?

Preprocessor that adds XML syntax to JavaScript.

Takes XML input and produces native JavaScript.

Using JSX

Objectives

- Write JSX
- Use React with JSX
- Use React without JSX

JSX is not exactly HTML

- You can use HTML entities within JSX text
  - `<div>&copy; all rights reserved</div>`
- React will not render properties on HTML elements that don't exist in the HTML spec.
  - preface custom properties with data-
    - `<div data-custom-attribute="foo" />`
  - Arbitrary attributes are supported on custom elements
    - `<x-my-component custom-attribute="foo" />`

JSX is not exactly HTML (cont.)

- XML syntax required
  - Elements must be closed
- Attributes use DOM property names
  - `className` instead of `class`
- Attributes become props in the child
- React components start with upper-case
- HTML tags start with lower-case

Using React with JSX

```
class LoginBox extends React.Component({
render() {
  return (
  <div>
  <label>Log In <input type="text" id="username"
      placeholder={this.props.placeholderText} />
  </label>
  </div>
  );
}
});
ReactDOM.render(<LoginBox placeholderText="Enter
Your Email" />, document.getElementById("login"));
```

React.createElement

```
createElement(
  string/ReactClass type,
  [object props],
  [children …]
)
```

- Create and return a new ReactElement of the given type.
- Type argument can be an HTML tag name string (`'div'`, `'span'`, etc.), or a ReactClass (created via `React.createClass`).
- JSX gets compiled to React.createElement

Using React without JSX

```
...
  render() {
    return React.createElement("div",null,
      React.createElement("label",null,"Log In",
        React.createElement("input",
         { type: "text", id: "username" })
      )
    );
  }
});
ReactDOM.render(React.createElement(LoginBox, null),
document.getElementById("login"));
```

Expressions in JSX

- Use curly braces around JavaScript expressions to include them in JSX.

```
return(
  <div>
    <HelloWorld
    name={this.props.firstname +
      this.props.lastname} />

    {isLoggedIn ? <Logout /> : <Login />}
  </div>
  );
```

Expressions in JSX

```
class Logout extends React.Component{
  render() {
    return (<div>logout</div>);
  }
}

class Login extends React.Component{
  render(){
    return(<div>login</div>);
  }
}

class Container extends React.Component{
  render() {
    return(
    <div>
    {this.props.isLoggedIn ? <Logout /> : <Login />}
    </div>);
  }
}

ReactDOM.render(<Container isLoggedIn={false} />, document.getElementById("app"));
```

Precompiled JSX

- Two ways to use JSX
  - Compile to JavaScript during build
    - Use Babel
  - Serve JSX and compile in the browser
    - Use JSXTransformer
      - Intended only for prototypes
      - Deprecated

React Components

Objectives
- Understand component life-cycle
- Use events and dispatching
- Communicate between components
- Test React components

Creating Components

- Two techniques
  - `React.createClass`
  - `React.Component`
- `React.Component` is the ES6 way.
- `React.createClass` is deprecated as of v15.5.0.

`React.createClass`

```
createClass(object specification)
```

- Creates a component, given a specification.
- Implements a `render` method
  - `render` method is required.
    - `render` returns one single child.
      - Can be a virtual representation of a DOM component or another component that you've created
        - » may have a deep child structure
- Is still available for React 16+ as create-react-class

React.Component

- Base class for React Components when defined using ES6 classes.

```
class HelloMessage extends React.Component {
  render() {
    return (<div>Hello {this.props.name}</div>);
  }
}
ReactDOM.render(<HelloMessage name="Ann" />, mountNode);
```

F.I.R.S.T.

- React Components should be:
  - **F**ocused
  - **I**ndependent
  - **R**eusable
  - **S**mall
  - **T**estable

Single Responsibility

- A component should only do one thing.
- If it ends up growing, it should be decomposed into smaller subcomponents.
- A responsibility is a "reason to change."
- Single responsibility makes components more robust.

"A class should have only one reason to change.

-*Robert C. Martin*

Pure Functions

- Pure functions always return the same result given the same arguments.
- Pure function's execution doesn't depend on the state of the application.
- Pure functions don't modify the variables outside of their scope (no side effects).

Benefits of Pure Functions

- Easy to test
- Easy to reason about
- Easy to reuse
- Easy to reproduce the results

Function Comparison

slice()

```
var toppings =
['cheese','pepperoni','mushrooms'];

toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
```

- Always returns the same result given the same arguments
- Doesn't depend on the state of the application
- Doesn't modify variables outside its scope
- **It's a Pure Function!**

splice()

```
var toppings =
['cheese','pepperoni','mushrooms'];

toppings.splice(0,2);
// ["cheese", "pepperoni"]
toppings.splice(0,2);
// ["mushrooms"]
toppings.splice(0,2);
// []
```

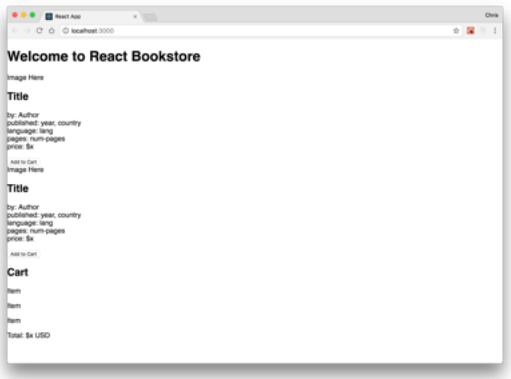
- **Not Pure!**

Stateless Functional Components

- If your component only has a render method and optional props, you can just create a normal JavaScript function.

```
function HelloWorld(props){
    return (
        <p>Hello {props.name}</p>
    );
}
reactDOM.render(<HelloWorld name='Chris' />,
document.getElementById("app"));
```

Lab 05: Static Version



Styles in React

- Facebook recommends using inline styles, set using JavaScript.
- Pass styles into JSX using objects containing style properties.
- Properties are camelCased versions of the CSS properties.

```
var headingStyle = {
  color: "blue",
  textTransform: "uppercase"
};
return (<h1 style={headingStyle}>
        Welcome!</h1>);
```

Styles in React

- Other ideas
  - CSS Modules
    - Create separate style modules and import and use them in components.
  - Use a CSS library (Bootstrap) for layout, inline for presentation.
- How to do CSS in React is still very much being worked out.
  - Lots of opinions

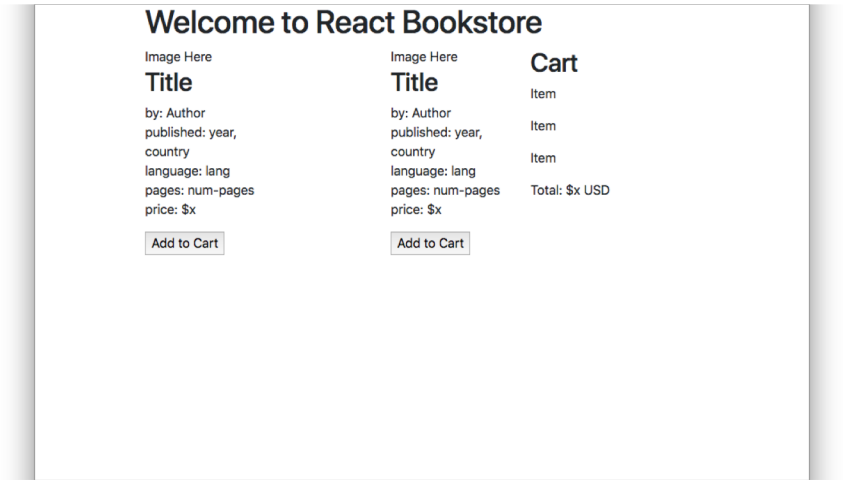
Styled Components

1. import styled from 'styled-components';
2. Call styled.[object] function, passing in style info using Tagged Template Literal Notation.

```
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

<Title>This is a styled component</Title>
```

Lab 06: Styling React



Context API

- Context allows parents to pass data implicitly to children, no matter how deep the component tree is.

Render Props

- Allows you to use props to share code between two components.

```
class App extends React.Component {
  getChildContext() {
    return {
      color: 'red'
    }
  }
  render() {
    return <Button />
  }
}
App.childContextTypes = {
  color: React.PropTypes.string
}
```

```
// Hook 'Color' into 'App' context
class Color extends React.Component {
  render() {
    return this.props.render(this.context.color);
  }
}

Color.contextTypes = {
  color: React.PropTypes.string
}

class Button extends React.Component {
  render() {
    return (
      <button type="button">
        {/* Return colored text within Button */}
        <Color render={ color => (
          <Text color={color} text="Button Text" />
        ) } />
      </button>
    )
  }
}

class Text extends React.Component {
  render(){
    return (
      <span style={{color: this.props.color}}>
        {this.props.text}
      </span>
    )
  }
}

Text.propTypes = {
  text: React.PropTypes.string,
  color: React.PropTypes.string,
}
```

Using Context with Provider

```
const ColorContext =
React.createContext('color');
class ColorProvider extends
React.Component {
    render(){
        return (
        <ColorContext.Provider
             value={'red'}>

         { this.props.children }

         </ColorContext.Provider>
        )
    }
}
```

```
class Parent extends React.Component
{
    render(){
        return (
            <ColorProvider>
                <Child />
            </ColorProvider>
        );
    }
}
```

Composition

- Composition is combining smaller components to form a larger whole.

Reusable Components

- Break down the common design elements (buttons, form fields, layout components, etc.) into reusable components with well-defined interfaces.
- The next time you need to build some UI, you can write much less code.
- This means faster development time, fewer bugs, and fewer bytes down the wire.

Container Components

- Wrap presentational components
- Contain the logic and state

Presentational Components

- Components that just output presentation
- Contain no logic

props

- Props are passed from the parent component to its children
- JSX attributes become a single object in the child.

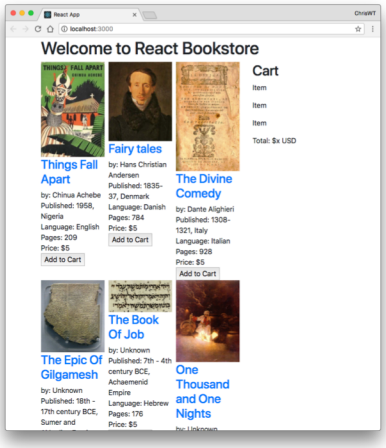
- Parent:

```
<Hello name="Chris" />
```

- Child:

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

Lab 07: Props and Containers



Step 3 - Minimal UI State

- Is it state?
  - Is it passed from the parent via props?
    - Probably not state
  - Does it change over time?
    - Might be state
  - Can you compute it based on any other state or props in your application?
    - Probably not state

Props vs. State

Props

- Passed to the child within the render method of the parent
- Immutable
- Better performance

State

- State of the parent becomes prop of child
- Mutable

Setting Initial State

```
class MyComponent extends React.Component {
  constructor() {
      this.state = { /* some initial state */ }
  }
```

super()

- Used for calling methods on the parent.
- Makes `this` available in the constructor.
- ES6 classes must call `super()` if they are subclasses.
- If you don't have a constructor, you don't need to call `super()`. React will automatically make `this.props` available to the subclass.
- If you want to use `this.props` in your constructor, you need to call `super(props)` in the constructor.

Using super()

```
class App extends React.Component {
  render() {
      return(
         <GameContainer
            game="Pac-Man"
            username="Chris" />
      );
   }
}
```

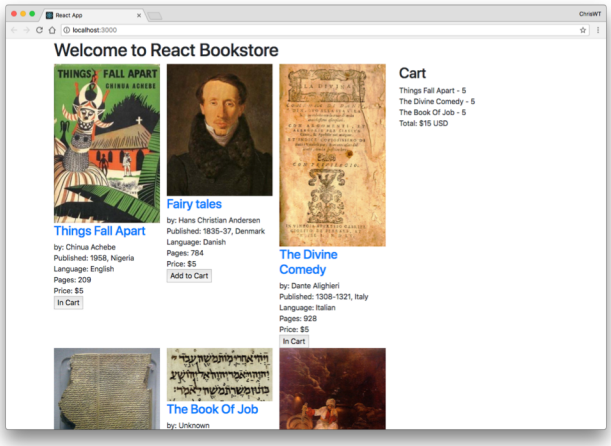
Using super() (cont.)

```
class GameContainer extends React.Component {
  constructor(props) {
      super(props);
      this.state = {
         playerName: props.username + " The Great"
      }
  }
   render() {
      return(<div>Welcome to
            {this.props.game},
            {this.state.playerName}</div>);
   }
}
```

Step 4 – Where Should Your State Live?

- For each piece of state:
  - Identify every component that renders something based on that state.
    - Find a common owner component
    - The common owner or a component higher in the hierarchy should own the state.
    - If you can't find a common owner, create a new component higher up in the hierarchy just for holding the state.

Lab 08: Adding State



Step 5 – Add Inverse Data Flow

- Use a function to update state in components higher up in the hierarchy.

reactjs.org/hooks

HOOKS

Why Use Hooks?

- Stateful classes need to be written with class syntax
- Classes can be confusing.
- Reusing logic is not possible

Stateful Components with Class

```
Import React from 'react';

export default class Hello extends React.Component {
      constructor(props) {
            super(props);
            this.state = {
                  name: 'World',
            }
            this.handleChange = this.handleChange.bind(this);

      }
      handleNameChange(e) {
            this.setstate({
                  name: e.target.value
            });
      }
      render() {
            return(
                  <input type="text"
                          value={this.state.name}
                          onChange = {this.handleChange} >
            );
      }
}
```

What are Hooks?

- Functions provided by React that let you hook into React features from function components.
- Must be at the top level of the component
  - not inside a condition
- Hook examples
  - useState – Hooks into state
  - useContext – Hooks into Context
  - useEffect – Hooks into lifecycle methods. Runs after initial render and after every update (by default).

useState Hook

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

export default function Hello(props) {
   const [name, setName] = useState('World');

   function handleChange(e) {
      setName(e.target.value);
   }

   return(
      <input type="text"
             value={name}
             onChange={handleNameChange} />
   );
}
```

useContext Hook

```
import React, { useState, useContext } from 'react';
import { ThemeContext } from './context';

export default function Hello(props) {
    const theme = useContext(ThemeContext);
    const [name, setName] = useState('World');

    function handleChange(e) {
        setName(e.target.value);
    }
    return(
        <section className={theme}>
            <input type="text"
                   value={name}
                   onChange={handleChange}
            />
        </section>
    )
}
```

useEffect Hook

```
import React, { useState, useEffect } from 'react';

export default function Hello(props) {
    const [name, setName] = useState('World');

    useEffect(() = {
        document.title = 'Hello' + ' ' + name;
        return ()=>{
            // optionally return a function
            // to clean up after the effect,
            // such as by removingEventListener or timer set
            // in the effect
    });

...
```

Custom Hooks

- Start with 'use' by convention.
- Create a function outside of the component function.
- Use the function inside the component.

Custom Hook Example

```
export default function Hello(props) {
   const [name, setName] = useState('World');
   useDocumentTitle(name);
   ...
}

function useDocumentTitle(title){
   useEffect(() = {
   document.title = title;
   }
}
```

Should you convert everything to Hooks?

- Probably not right now (Q1 2019).
- Hooks are not yet released (as of 16.7).
- You can try out hooks now with the alpha version.

Forms

Objectives

- Use Controlled Components
- Use Uncontrolled Components

Forms Have State

- They are different from other native components because they can be mutated based on user interactions.
- Properties of Form components
  - value
    - supported by `<input>` and `<textarea>`
  - checked
    - supported by `<input type="checkbox | radio" />`
  - selected
    - supported by `<option>`
- `<textarea>` should be set with value attribute, rather than children in React

Form Events

- Form components allow listening for changes using onChange
- The onChange prop fires when:
  - The value of `<input>` or `<textarea>` changes.
  - The checked state of `<input>` changes.
  - The selected state of `<option>` changes.

Events

- SyntheticEvent
  - A cross-browser wrapper around the browser's native event
  - Same interface as browser's native event

```
class ShareButton extends Component ({
  onButtonClick (evt) {
    alert("wow!");
  }
  render () {
    return (
    <div onClick={this.onButtonClick}>Share!</div>
    );
  }
});
```

Controlled Components

- A controlled `<input>` is one with a value prop.
- User input has no effect on the rendered element.
- To update the value in response to user input, you can use the onChange event.
- Controlled vs. Uncontrolled Demo:
  - https://goo.gl/wMMbVc

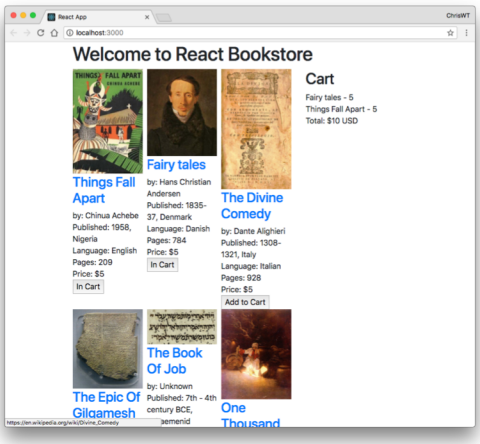
Uncontrolled Components

- An `<input>` without a value property is an uncontrolled component.

```
render() {
  return <input type="text" />;
}
```

- User input will be reflected immediately by the rendered element.
- Maintains its own internal state

Lab 09: Interactions, Events, Callbacks



Component Life-Cycle Events

- 2 Categories
  - Mount / Unmount
  - When component receives new data

Life-Cycle Methods

- Methods of components that allow you to hook into views when specific conditions happen.

139

Mount/Unmount

- Mount and unmount methods are called when components are added to the DOM (Mount) and removed from the DOM (Unmount).
- Each is invoked only once in the lifecycle of the component
- Used for:
  - establish default props
  - set initial state
  - make AJAX request to fetch data for component
  - set up listeners
  - remove listeners

140

Mount/Unmount Life-Cycle Methods

- `constructor`
- `componentDidMount`
- `componentWillUnmount`

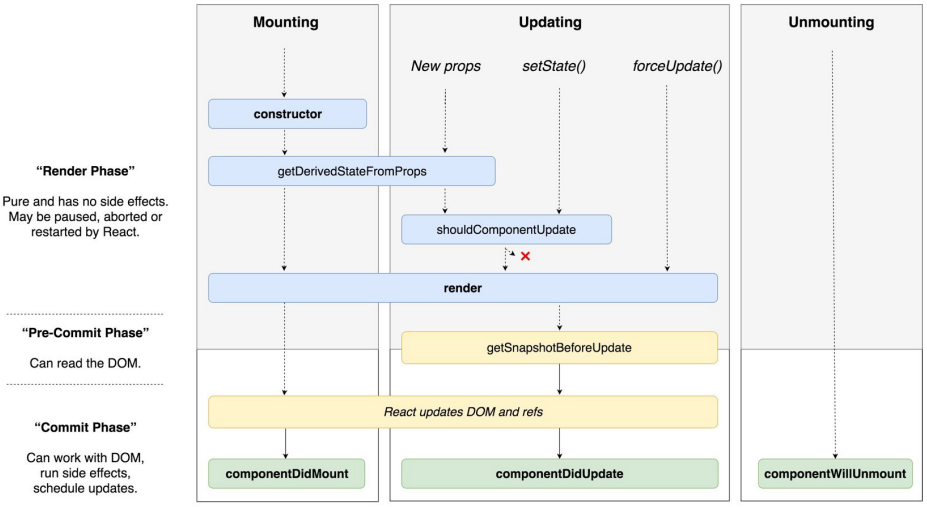
141

Data Life-Cycle Methods

- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `getSnapshotBeforeUpdate`
- `componentDidUpdate`

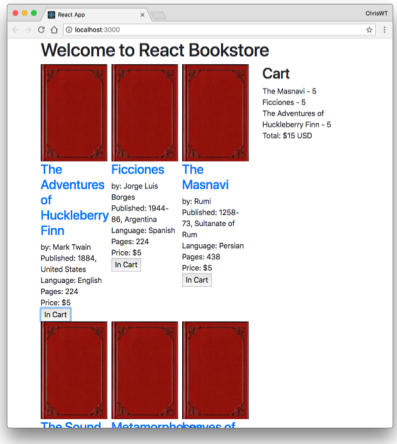
142

Component Life Cycle



143

Lab 10: Component Life-Cycle



Testing React Components

Objectives

- Learn about different rendering modes
- Learn about Jest
- Write Unit Tests with Jest

What to Test in a React Component

- Does it render?
- Does it render correctly?
- Test every possible state / condition
- Test the events
- Test the edge cases

PropType

- Allows you to add type to props
- Useful for debugging, documentation
- Types:
  - `array`
  - `object`
  - `string`
  - `number`
  - `bool (not boolean)`
  - `func (not function)`
  - `node`
  - `element`

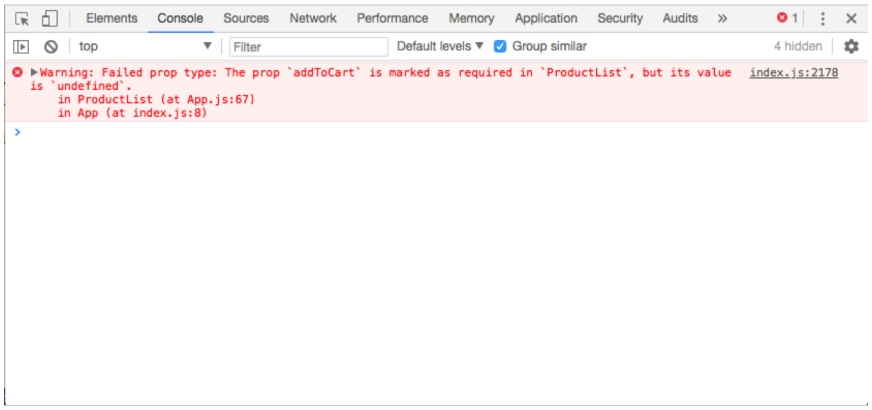
Using PropTypes

```
import React from 'react';
import PropTypes from 'prop-types';

class Component extends React.Component {
...
}

Component.propTypes = {
    name: PropTypes.string.isRequired,
    size: PropTypes.number.isRequired,
    color: PropTypes.string.isRequired,
    style: PropTypes.object
};
```

Lab 11: PropTypes



Jest

- Facebook's Testing Framework
- Runs any tests in __tests__ directories, or named .spec.js, or named .test.js
- Simulates browser environment with jsdom

Mocking

- Mock Function - erase the implementation of a function
- Manual Mocking - stub out functionality with mock data
- Timer Mocking - swap out native timer functions

Mock Function

```
const mockCallback = jest.fn();
forEach([0, 1], mockCallback);

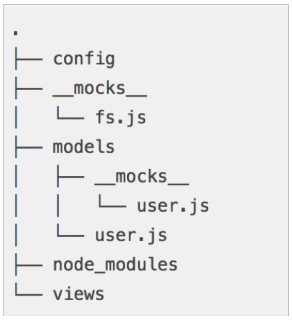
// The mock function is called twice
expect(mockCallback.mock.calls.length).toBe(2);

// The first argument of the first call to the function was 0
expect(mockCallback.mock.calls[0][0]).toBe(0);

// The first argument of the second call to the function was 1
expect(mockCallback.mock.calls[1][0]).toBe(1);
```

Manual Mock

- Ensures tests will be fast and reliable by mocking external data and core modules.
- Define in a __mocks__ subdirectory adjacent to the module



Manual Mocks (cont.)

- Recommended practice is to create an automatic mock and then override it.
- See https://github.com/facebook/jest/tree/master/examples/manual_mocks for examples of manual mocks and tests that use them.

Automocking

- Disabled by default, but can be used on a per-module basis
  - `jest.mock(moduleName)`
- Enable automock by default in jest.config
  - `automock:true`

Snapshot Testing

1. Renders a components
2. Creates a "snapshot file" on first run
3. Compares subsequent runs with first and fails test if different.

Sample Snapshot Test

```
import React from 'react';
import Link from '../Link.react';
import renderer from 'react-test-renderer';

it('renders correctly', () => {

const tree = renderer.create( <Link
   page="http://www.facebook.com">Facebook</Link>
   ).toJSON();

expect(tree).toMatchSnapshot(); });
```

TestUtils

- `renderIntoDocument()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDomComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `more...`
- **NOTE:** TestUtils has been moved into react-dom as of v15.5.0

TestUtils Example

```
Old (pre-15.5.0):
import TestUtils from 'react-addons-test-utils';

New (15.5.0):
import TestUtils from 'react-dom/test-utils';
```

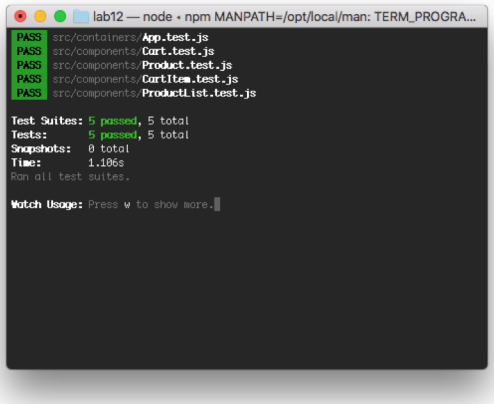
Enzyme

- Testing utility for React, created by Airbnb
- 3 render modes
  - shallow
    - render just the component
  - mount (Full Rendering)
    - For testing where you have components that require the full lifecycle in order to test.
    - Need to run in a browser environment.
  - render
    - renders react components to static HTML
- To use, pass a React component into a mode method.
  - `const wrapper = shallow(<PollSubmitButton />);`

Shallow Rendering

- Renders just a single component, regardless of where it is in the hierarchy.
- Allows you to isolate components for testing.
  - Ensure that your tests are indirectly asserting on behavior of child components.

Lab 12: Testing with Jest and Enzyme



Flux and Redux

Objectives

- Understand the Flux pattern
- Explain Redux's architecture
- Create Redux actions
- Write pure functions
- Use Reducers
- Use Redux with AJAX

Flux

- Flux isn't a library or module.
- It's a design pattern.
- npm install flux installs Facebook's dispatcher.
- It's possible to use Flux design principles without Facebook's module.

Flux Flow

1. Some sort of interaction happens in the view.
2. This creates an action, which the dispatcher dispatches.
3. Stores react to the dispatched action if they're interested, updating their internal state.
4. Stateful view component(s) hear the change event of stores they're listening to.
5. Stateful view component(s) ask the stores for new data, calling setState with the new data.

Flux Action

- An action in flux is what's made when something happens.
- In other words, when you click on something, that's not an action
  - it creates an action. Your click is an interaction.
- Actions should have (but aren't required to have) a type and a payload. Most of the time they will have both, occasionally they'll just have a type.

Flux Dispatcher

- Broadcasts actions when they happen and it lets things tune in to those broadcasts
- Instead of an onClick function using a callback passed to it to set the state of your application, you have it (onClick) use the dispatcher to dispatch a specific action for anyone who's interested to listen for.

Flux Stores

- Represents the ideal state of your application
- If a user enters something into a form, it dispatches an action. If the store is listening for this action, it will update its internal state accordingly.
- Stores don't contain any public setters, just public getters. The only ones who can change the data in a store is the store itself when it hears an action from the dispatcher that it's interested in.

EventEmitter

- Stores emit change events that don't contain data.
- If the view is listening for the particular store's change event, the view should ask the store for the new data that will bring the view back into sync, call setState, and re-render.

Redux

- An implementation of Flux
- Stores state of the app in an object tree in a single store
- The state tree can only be changed by emitting an action
- Specify how the actions transform the state tree using pure reducers

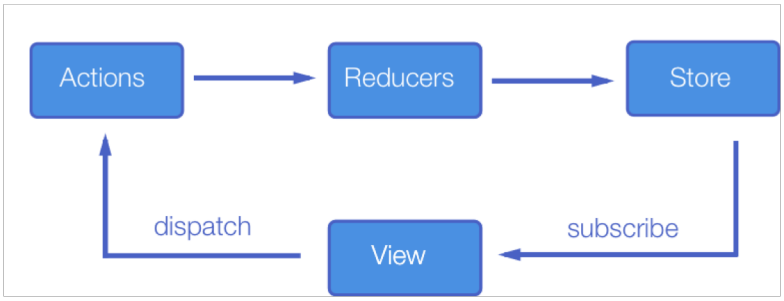
Stores & Immutable State Tree

- The biggest difference between Flux and Redux is that Redux has a single store.
- Redux has a single store with a single root reducing function.
- Split the root reducer into smaller reducers to grow the app.
- Trace mutations by replaying actions that cause them.

Redux Actions

- Payloads of information that send data from the application to the store.
- Actions are the only way to mutate the internal state.
- Use `store.dispatch()` to send them to the store.

```
const ADD_TODO = 'ADD_TODO'
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```



Reducers

- Specifies how the application's state changes in response to something happening (an action).
- A pure function that takes the previous state and an action, and returns the next state.
- `(previousState, action) => newState`

Things You Should Never do in a Reducer

- Mutate its arguments
- Perform side effects like API calls and routing transitions
- Call non-pure functions

Reducer Composition

- The fundamental pattern of building Redux apps
- Split up reducer functions using child reducers.

- Workflow
  - Write top-level reducer to handle a particular function.
  - Break up the top-level reducer into a master reducer that calls smaller reducers to separate concerns.

Reducer Composition Example

```
function checkedValue(state = [], action) {
  switch (action.type) {
      case 'SELECT_ANSWER':
        return state
          .slice(0,action.index)
          .concat([action.value])
          .concat(state.slide(action.index+1));
        default:
           return state;
    }
}
export default checkedValue;

function question(state = [], action) {
  return state;
}
export default questions;
```

Reducer Composition Example (cont.)

Combine reducers

```
const rootReducer = combineReducers({
  questions,
  checkedValue
});

Export default rootReducer;
```

Redux Store

- Holds the application state
- Allows access to state via `getState()`
- Allows state to be updated via `dispatch(action)`
- Registers listeners via `subscribe(listener)`
- Handles unregistering of listeners via the function returned by `subscribe(listener)`

Redux Pros and Cons

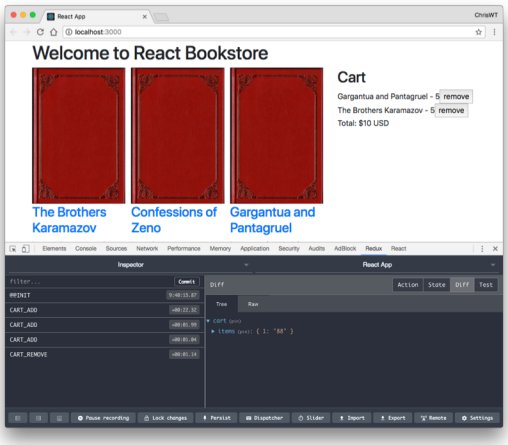
Redux Pros

- Declarative
- Immutable state
- Mutation logic separate from views
- Great for testing

Redux Cons

- More complicated than just using plain react components

Lab 13: Implementing Redux



What is Redux Middleware?

- a third-party extension point between dispatching an action and the reducer.

What is Middleware Good For?

- Logging actions
- Reporting errors
- Dispatching new actions
- Asynchronous requests

React AJAX Best Practices

- Four Ways
  - Root Component
    - Best for small apps and prototypes.
  - Container Components
    - Create a container component for every presentational component that needs data from the server.
  - Redux Middleware
    - Thunk Middleware
    - Saga
  - Relay
    - Good for large applications
    - Declare data needs of components with GraphQL
    - Relay automatically downloads data and fills out the props

Using React with Other Libraries

- Use the lifecycle events to attach logic from other libraries.
- `componentDidMount`
  - Attach code that will run when the component is loaded.
  - Can be used similarly to jQuery's $(document).ready()
- `componentDidUpdate`
  - Attach code that will run when a component updates.

Redux Thunk

- Allows you to write action creators that return a function instead of an object.

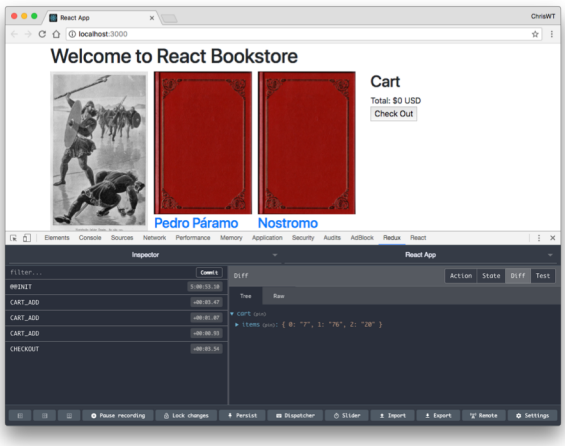
How is Thunk Useful?

- delay the dispatch of an action
- dispatch only if a certain condition is met
- perform async network operation
  - database
  - AJAX

How does Thunk work?

1. Redux Thunk middleware is added to the store
2. When an action creator returns a function, that inner function will get executed by the Thunk middleware.
3. The inner function receives the store methods `dispatch` and `getState()` as parameters.
4. On success, the Thunk action calls a standard action.

Lab 14: Thunk



Redux Saga

- Useful for complex async operations.
- Uses generator functions to complete actions in series.
- redux-saga.js.org/docs/introduction/BeginnerTutorial.html

Using Sagas

```
//install redux-saga
npm install --save redux-saga

//dispatch an action
class UserComponent extends React.Component {
...
onSomeButtonClicked() {
   const { userId, dispatch } = this.props;
   dispatch({
      type: 'USER_FETCH_REQUESTED',
      payload: {userId}});
   }
... }
```

Using Sagas (cont.)

```
// create a "worker" saga to do the async action

import { call, put, takeEvery, takeLatest } from 'redux-saga/effects';
import Api from '...';

// worker Saga: will be fired on USER_FETCH_REQUESTED actions

function* fetchUser(action) {
  try {
    const user = yield call(Api.fetchUser,
action.payload.userId);
    yield put({type: "USER_FETCH_SUCCEEDED", user: user});
} catch (e) {
    yield put({type: "USER_FETCH_FAILED", message:
e.message});
}
}
```

Using Saga (cont.)

```
// create a listener saga to listen for the event

function* mySaga() {
   yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/* Alternatively you may use takeLatest.

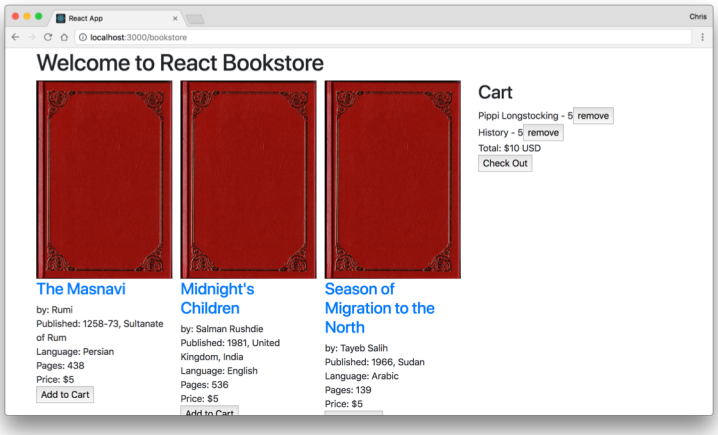
function* mySaga() {
   yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}
*/
// export mySaga
export default mySaga;
```

Using Saga (cont.)

```
// Connect Saga to the store, using middleware
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import reducer from './reducers';
import mySaga from './sagas';

// create the saga middleware
const sagaMiddleware = createSagaMiddleware();
// mount it on the Store
const store = createStore( reducer,
applyMiddleware(sagaMiddleware) );
// then run the saga
sagaMiddleware.run(mySaga);
// render the application
```

Lab 15: Persisting Data in localStorage



Advanced Topics

Objectives

- Server-side React
- Using React with Other Frameworks / Libraries
- Performance Issues

React Router

- Declarative way to do routing
- Maps components to URLs
- Dynamic routing (as of v4)
  - Routing takes place as the component is rendering
  - Not in a configuration
  - Almost everything in React Router is a component

Using React Router

- Import the version of React Router for your target environment (i.e. DOM or Native), plus other components

```
import { BrowserRouter, Route, Link } from 'react-router-dom'
```

- Render the Router

```
ReactDOM.render((
<BrowserRouter> <App/> </BrowserRouter> ), holder)
```

- Use <Link> to link to a new location (in <App> in this case)

```
<Link to="/dashboard">Dashboard</Link>
```

- Render a Route (also in <App>)

```
<Route path="/dashboard" component={Dashboard}/>
```

Router Rendering Example

index.js

```
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), holder)
```

- Use BrowserRouter when you have a server
- Use HashRouter if you're using a static file server.

Route Matching

- Route
  - Compares the value of the path prop to the current location's pathname.
  - Renders the component specified by the component prop.
  - Can be used anywhere you want to render based on location.
- Switch
  - Can be (optionally) used for grouping Routes. Will iterate through a group and stop when a match is found.

Navigation

- <Link>
  - Creates links in your application.
  - Inserts an <a> in your HTML
- <NavLink>
  - Can be styled as "active" when it matches the current location.
- <Redirect>
  - Forces navigation

Server-side React

- Allows you to pre-render components' initial state on the server
- Methods:
  - `renderToString`
    - Returns an HTML string
    - Send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.
  - `renderToStaticMarkup`
    - Works the same as `renderToString`, but doesn't add in the extra React DOM elements
    - Good for using React as a static page generator

Relay and GraphQL

Objectives

- Retrieve data with Relay

What is Relay?

- Framework for building data-driven react applications
- Simpler than Flux, but may also be used with Flux
- Use GraphQL as a query language.

GraphQL

- Data query language and runtime
- Declarative
- Compositional
- Strongly Typed

GraphQL Example

Query

```
{
  user(id: 3500401) {
    id,
    name,
    isViewerFriend,
    profilePicture(size:
50)  {
      uri,
      width,
      height
    }
  }
}
```

Response

```
{
  "user" : {
    "id": 3500401,
    "name": "Jing Chen",
    "isViewerFriend": true,
    "profilePicture": {
      "uri":
"http://someurl.cdn/pic.jpg"
,
      "width": 50,
      "height": 50
    }
  }
}
```

Relay Pros and Cons

Relay Pros

- Even more declarative
- No custom getter logic
- Tight server integration

Relay Cons

- Requires GraphQL server
- Much more complexity
- Less flexible

Performance Optimization

- Production bundle
- Perf object
- shouldComponentUpdate()

Development vs. Production

- Development build
  - Uncompressed.
  - Displays additional warnings that are helpful for debugging.
  - Contains helpers not necessary in production
  - ~669kb
- Production build
  - Compressed.
  - Contains additional performance optimizations.
  - ~147kb

Perf Object

- Indicates expensive parts of your app.
- Only available in development mode.

Perf Object Methods

- Methods
  - `start()` and `stop()`
    - Start and stop measurement
    - Records operations in-between
  - `printInclusive(measurements)`
    - Prints the overall time taken
    - Defaults to the measurements from the last recording
    - Outputs a nice-looking table
  - `printExclusive(measurements)`
    - Doesn't include time taken to mount components, process props, etc.
  - `printWasted(measurements)`
    - "Wasted" time is spent on components that didn't render anything
  - `printOperations(measurements)`
    - Prints the underlying DOM manipulations

Optimization Techniques

- Use `shouldComponentUpdate` hook to add optimization hints to React's diff algorithm

```
shouldComponentUpdate: function() {
   // Let's just never update this component again.
   return false;
 },
```

- Use keys
  - Unique identifier created with `key` attribute.

Using Pre-built Components

- Thousands of React components are shared via npm
  - Searchable through databases like React-Components.com.

Further Study

- Dan Abramov's 30 Free Redux Videos
- Wes Bos's Redux Course (https://learnredux.com/)

Where to go for help?

- React Documentation
- Stackoverflow

Disclaimer and Copyright