Kevin Chen

4/25/18

React Essential Training Notes

**Introduction**

Setting up Chrome tools for React

* Add the “React Developer Tools” and “react-detector” Chrome extensions.
  + React Developer Tools lets you view React components
  + React-detector detects whenever you go onto a React site.

Inspecting React sites

* Navigate to a site using React, such as Airbnb. Then open the developer tools, and the last tab should be called “React”. Open the “React” tab to see all of the React components on the webpage.
* You can click “Inspect” on a particular component to jump to that component in the React components tab.

**What is React.js?**

What is React?

* A JS library used to create UIs. Create reusable components, which display data as it changes over time.
* Created at Facebook and Instagram.
* React expanded to React Native, which allows you to create native mobile apps with React.

Efficient rendering with React

* React makes updating the DOM faster by using DOM Diffing.
  + DOM Diffing compares rendered content with the new UI changes
  + It makes only the minimal changes necessary.
  + It compares JavaScript objects, which is faster than writing to or reading from the DOM (e.g. getElementById()).

**Intro to JSX and Babel**

Pure React

* Write React code inside JS files that will modify your DOM.
* Create element:

const [variable] = React.createElement(

‘[type]’,

{[property1]: ‘[value1]’, [property2]: ‘[value2]’, …}

[Child elements]

)

* + Example (creates an h1 element with text “Hello World”, setting its id and class):

const title = React.createElement(

‘h1’

{id: ‘title’, className: ‘header’},

‘Hello World’

)

* We then need to render it:

ReactDOM.render(

[variable]

[DOM element]

)

* + Example (renders the title by targeting the element with id “react-container”

ReactDOM.render(

title,

document.getElementById(‘react-container’)

)

* React pages need to include two script files in the header (React and ReactDOM). Find them at cdnjs.
* To serve up your files, you can simply use httpster.
  + Install with the command “sudo install -g httpster”
  + Change to the directory containing the folder with the web page files, and launch the server with the command “httpster -d ./[folder with files] -p [port]”
* You can shorten/clean up the commands using ES6 destructuring:
  + const { [method] } = [package] lets you get rid of the package name when using the method.
  + Examples:

const { createElement } = React lets you shorten React.createElement() to createElement().

const { render } = ReactDOM lets you shorten ReactDOM.render() to render().

Refactoring elements using JSX

* Create a CSS element (example):
  + const style = {

backgroundColor: ‘orange’

color: ‘white’

fontFamily: ‘verdana

}

* Then use the CSS property in the style attribute of the HTML element (example continued):
  + const title = React.createElement(

‘h1’

{id: ‘title’, className: ‘header’, style: style},

‘Hello World’

)

* JSX is Javascript as XML: a tag based syntax to create react elements.
  + It just provides syntactic sugar for the React.createElement() function.
* Example:

render(

<h1 id=‘title’

className=‘header’

style={style}

Hello World (children of this element).

</h1>

)

* + {style} is a JSX expression.
  + Note, this will result in an error when you run, as it is not natively supported by the browser. You need to use a tool called Babel (more details below).
* We can use an in-line expression instead of variable (example continued):
  + style={style} can be replaced with:

style={{backgroundColor: ‘orange’, …}}

* + The outer set of brackets shows this is an expression. The inner set of brackets is an object.

Babel inline transpilling

* Babel is a JavaScript compiler. It takes in code that isn’t supported by the browser and returns code that is browser supported.
* Fastest way to use Babel is to use the browser transpiler. Download it from cdnjs.
  + Must use Version 5.x.x to use in-browser trnaspiler.
* Add a script tag in the head for Babel (specifying the source to Babel).
* Modify the body JavaScript script tag to be of type “text/babel”.
  + This indicates we should run the Babel in-browser transpiler on it.

Babel static transpilling with Babel-CLI

* For production, you want to handle transpiling before your scripts get to the browser.
  + This results in better performance because transpiling happens before runtime.
* Run “npm init” to generate the project, and fill out as many of the fields as you can.
  + This will create a package.json file.
* Run “npm install babel-cli@[version number] --save-dev”. (e.g. 6.18.0)
* Run as admin (or prefix sudo): “sudo npm install -g babel-cli”
  + This will let you run the Babel CLI command from anywhere on the computer.
* One recommended file structure:
  + src folder for React JSX code, and other/later features.
  + dist folder for code that is ready for the browser.
* Create a file with name “.babelrc”
  + We will setup the presets (i.e. everything we want to transpile using Babel) in this file.
  + Example:

{

‘presets’: [‘latest’, ‘react’, ‘stage-0’]

}

* Install the presets with the commands:
  + “npm install babel-preset-react@[version number] ---save-dev”.
  + “npm install babel-preset-latest@[version number] ---save-dev”.
  + “npm install babel-preset-stage-0@[version number] ---save-dev”.
  + Recommend 6.16.0 for version number.
* Now we can actually transpile our code:
  + Run “babel [file to transpile] --out-file [file to output transpilation]”
  + Example: “babel ./src/index.js --out-file ./dist/bundle.js”
* For convenience, you can use package.json to make the command to start up the web server shorter by adding the following in the “scripts” json object.

“scripts”: {

“start”: “httpster -d ./dist -p 3000”

}

* + Now you just need to type in “npm start” in the terminal.

Building with webpack

* Another approach to build JSX is to use module bundlers like Webpack.
* Webpack helps create static files and automates processes that need to happen before files can go into production.
* Webpack and combine several script files (along with dependencies and even CSS) into one file, resulting in just one request.
* Create a Webpack config file (webpack.config.js)
  + This file will describe everything we want Webpack to do to our files to make them ready for production.
* Example Webpack config file:

module.exports = {

entry: “./src/index.js” //Index source file

output: { // Specifies where we want to output these files

path: “dist/assets”,

filename: “bundle.js”,

// The folder name where the bundled file(s) will reside

publicPath: “assets”

}

devServer: { // Enables automatic reload after any changes.

inline: true,

contentBase: ‘./dist’ //where our dist files are located.

port: 3000

}

module: {

loaders: [ // Tasks you want Webpack to perform

{ // For babel

test: /\.js$/ //Any file with js extension.

//Exclude node\_modules folder  
 exclude: /(node\_modules)/,

loader: [“babel-loader’]

query: {

presets: [‘latest’, ‘stage-0’, ‘react’]

}

}

]

}

}

* Add “var webpack = require(“webpack”)” at the beginning of the file.
* Run “npm install webpack@[Version 1] --save-dev”. Recommend Version 1.13.3.
* Run “npm install npm install [babel-loader@6.2.5](mailto:babel-loader@6.2.5) --save-dev”.
* Run “npm install [webpack-dev-server@1.16.2](mailto:webpack-dev-server@1.16.2) --save-dev”.
* Run as admin (or sudo): “npm install -g [webpack@1.13.3](mailto:webpack@1.13.3)” to install Webpack globally so that it can be run from anywhere.
  + Alternatively run webpack with command “[/path/to/node\_modules]/.bin/webpack]”
* Start the webpack dev-sever with ‘./node\_modules/.bi/webpack-dev-server”

Loading JSON with webpack

Adding CSS to webpack build

Migrating to webpack 3

**React Components**

Planning an ActivityCounter

* We are going to be creating a counter to keep track of the number of times we do a particular activity. Must have a location, date, and optionally have 2 boolean values.

Creating Components with createClass()

* Import React: “import React from ‘react’”
* Create a React component by creating a class (example):

// components/SkiDayCount.js

//export is required if you refer to this in another file.

export const SkiDayCount = React.createClass({

render() {

return (

<div className=“ski-day-count”>

<div className=“total-days”>

<span>5 days</span>

</div>

<div className=“powder-days”>

<span>2 days</span>

</div>

<div className=“backcountry-days”>

<span>1 hiking day</span>

</div>

</div>

)

}

})

* Now you can call this from some other place in the code. If you call it from another file, then you need to import it: “import { SkiDayCount } from ‘./components/SkiDayCount’
  + Example:

render(

<SkiDayCount />,

document.getElementById(‘react-container)

);

* Add window.React = React at the top of the file (below import statements) to get rid of errors that say “React is not defined”.
* Note: createClass() is not used/recommended as much as ES6 classes or stateless functional component.

Adding Component Properties

* Think of properties in React as objects. Each property is a key with an associated value.
* When using the component, you can specify properties that the component uses, letting you set values dynamically. Example:

render(

<SkiDayCount

total={50}

powder={20} //JSX expression.

backcountry={10}

goal={100}/>,

document.getElementById(‘react-container)

);

* + Don’t use JSX expression for String
* Now the component can use these properties like so (example):

<div className=“total-days”>

<span>{this.props.total} days</span>

</div>

Adding component methods

* You can create methods that are encapsulated in the component. Example:

export const SkiDayCount = React.createClass({

percentToDecimal(decimal) {

return ((decimal \* 100) + ‘%’)

},

calcGoalProgress(total, goal) {

return this.percentToDecimal(total/goal)

}

render() { … }

* You can then call the method elsewhere in the component:

render(

…

<span>

{this.calcGaolProgress(this.props.total, this.props.goal)}

</span>

Creating components with ES6 Class Syntax

* Instead of createClass(), extend the React.Component class. Example:

export class SkiDayCount extends React.Component{

percentToDecimal(decimal) {…}

calcGoalProgress(total, goal) {…}

render() {…}  
}

* Use more declarative syntax:
  + Example: choose what to import from React

Import { Component } from ‘react’

// Now just say Component instead of React.Component

Export class SkiDayCount extends Component {…}

Creating stateless functional components

* A third way to create components is to use a function.
* Stateless functional components are functions that take in property information and return JSX elements.
* Stateless components can’t access “this”, so properties are passed directly into the function. Local methods need to be removed and put into their own functions.
* Example:  
  // Component methods must be factored out.

// Can be used with any component.

const percentToDecimal = (decimal) => {

return ((decimal \* 100) + '%')

}

const calcGoalProgress = (total, goal) => {

return percentToDecimal(total/goal)

}

//The actual component:

export const SkiDayCount = (props) => (

<div className="ski-day-count">

<div className="total-days">

<span>{props.total}</span>

<span>days</span>

</div>

<div className="powder-days">

<span>{props.powder}</span>

<span>days</span>

</div>

<div className="backcountry-days">

<span>{props.backcountry}</span>

<span>days</span>

</div>

<div>

<span>

{calcGoalProgress(

props.total,

props.goal

)}

</span>

</div>

</div> ) // Parenthesis means we are returning some JSX elements.

* + Notice how “props” has been passed in and “this.props” is replaced with “props”
* We can destructure the properties so that we don’t have to use the whole object.

export const SkiDayCount = ({total, powder, backcountry, goal}) => (

…

<span> {total} </span> // Replaces props.total

… // Do similar thing for other div’s.

* Typically a good idea to use stateless components whenever possible.
  + Offers a functional way to work with components. May have some performance benefits.

Adding React-Icons

* There are a lot of community tools that people have built to supplement React.
* To install a community tool, run the command “npm install --save [tool-name]”
  + Example: “npm install --save react-icons”
* Then import the components you want from the tools.
  + Example: “import Terrain from ‘react-icons/lib/md/terrain’

**Props and State**

Composing components

* React user interfaces are just a collection of composed components.

Displaying child components

* You can loop over the elements of a list using the array.map() function, which takes in a callback function that has a parameter using each value of the array.
  + Simple example:

const numbers = [1, 2, 3, 4, 5]

const doubled = numbers.map((number) => number \* 2);

console.log(doubled); // logs [2, 4, 6, 8, 10]

* + Very useful for a table. Example:

<table>

<thead> … </thead>

<tbody>

{days.map((day, i) =>

<SkiDayRow key={i}

resort={day.resort}

date={day.date}

powder={day.powder}

backcountry={day.backcountry}

/>

)}

</tbody>

</table>

* + “days” is a list of objects. Each object contains a string (resort), date, and two boolean values (powder and backcountry).
  + A stateless functional component is created for each element in “days”.
  + Whenever rendering children from an array, you need to supply a key. (This helps with smart rendering, allowing React to know what to update.)
  + map
* The spread operator lets you concisely push all of the properties from the array object into the props object. Example (continued):

<SkiDayRow key={i}

{…day}/>

Note: the three dots in this example are three literal dots in the code. In all other examples (unless otherwise specified), three dots are used to truncate code to shorten the example.

Default Props

* We can use default values as properties if another value is not provided.
* Setting default values for components made by createClass():
  + Create a getDefaultProps() that returns an object with default values.
  + Example:

export const SkiDayCount = createClass({

getDefaultProps() {

return {

total: 50,

powder: 50,

backcountry: 15,

goal: 100

}

},

… );

* Setting default values for components using ES6 class syntax:
  + Add the default values to the class instance by adding lines below the class definition. Example:

export class SkiDayCount extends Component { … }

SkiDayCount.defaultProps = {

total: 50,

powder: 10,

backcountry: 15,

goal: 75

}

* Setting default values for components using the stateless functional components.
  + Use the exactly same code (and place it in the same location) as you would for an ES6 class. Refer to above example.
  + Another way is to set it in the properties declaration. Example:

export const SkiDayCount = ({total=70, powder=20,

backcountry=10, goal=100}) => ( … );

Validating with React.PropTypes

* Proptypes allow us to supply a property type for all the different properties so that it will validate to make sure that we’re supplying the right type.
* Import Proptypes: import { propTypes } from ‘react’
* Supplying the wrong PropType results in a warning in the browser console.
* Using PropTypes for components created by createClass(). Add a propTypes object. Example:

import { createClass, propTypes } from ‘react’

export const SkiDayCount = createClass({

propTypes: {

total: PropTypes.number.isRequired,

powder: PropTypes.bool,

backcountry: PropTypes.number,

},

…

});

* + isRequired requires that a value for the property is provided.
* Using PropTypes for components created using ES6 class syntax:
  + Add the PropTypes to the class instance example:

export class SkiDayCount extends Component { … }

…

SkiDayCount.propTypes = {

total: PropTypes.number,

powder: PropTypes.bool,

backcountry: PropTypes.number

}

* Setting PropTypes for components using the stateless functional components.
  + Use the exactly same code (and place it in the same location) as you would for an ES6 class. Refer to above example.

Custom Validation

* PropTypes has a date type. Example:

SkiDayCount.propTypes = {

date: PropTypes.instanceOf(Date)

}

* You can write your own function that does custom validation on an object. If there is an error with the type, you return a new instance of the class Error. If no error, return null. Example:

SkiDayList.propTypes = {

days: function(props) {

if(!Array.isArray(props.days)) {

return new Error("SkiDayList should be an array")

} else if(!props.days.length) {

return new Error("SkiDayList must have at least 1 record")

} else {

return null

}

}

}

Working with State

* State represents the possible conditions (or states) of your application
* Goal of identifying the minimal representation of app state and then reducing the state to as few components as possible.
* This helps avoid writing too many state variables.
* Create a component for each state.
* getInitialState() in a createClass() component is how you initialize the default state.
  + It returns an object of properties.

import { createClass } from 'react'

export const App = createClass({

getInitialState() {

return {

allSkiDays: [

{

resort: "Squaw Valley",

date: new Date("1/2/2016"),

powder: true,

backcountry: false

},

…

]

}

},

render() {

return (

<div className="app">

//Prints “Squaw Valley”.

{this.state.allSkiDays[0]["resort"]}

</div>

)

}

})

Passing state as props

* Use “this.state” to get the current state properties.
  + “this.state.allSkiDays” would access the state property named “allSkiDays”.
* Example 1:

const { allSkiDays } = this.state

//Equivalent to const allSkiDays = this.state.allSkiDays

* Example 2:

<SkiDayList days={this.state.allSkiDays}/>

State with ES6 Classes

* If you are using states with ES6 classes, you need to add a constructor method instead of using getInitialState().
* Example:

export class App extends Component {

constructor(props) {

super(props)

this.state = {

allSkiDays = [ … ]

… //Any other state properties.

}

}

… // Everything else for the class is the same.

}

**Using the React Router**

Incorporating the Router

* The React Router is a community tool to add navigation among pages on a React site.
* To install the router, run “npm install [react-router@3.0.0](mailto:react-router@3.0.0) --save”
* Add the following imports: “import { Router, Route, hashHistory } from ‘react-router’”
* Put the component you want to render for a particular path using the “component” property of the Link component.
* Put multiple route components for a single URL path if you want multiple components to be rendered in a single page.
* hashHistory lets the app keep track of the changes happening in the address bar of the browser.
* Example:

<Router history={hashHistory}>

//Routes homepage URL to the App component.

<Route path="/" component={App}/>

//Routes any other URL to a component that displays a 404 error.

<Route path="\*" component={Whoops404}/>

</Router>

Setting up routes

* We want our app to handle have more than just one page. We can route each page to the App component, but have the app component render different elements based on the URL.
* The App component can access the pathname using the code:

this.props.location.pathname

* This if-statement checks if you went to the homepage:

if (this.props.location.pathname === “/”) {

// Display components that are for the homepage.

}

Navigating with the link component

* The Link component lets you navigate between pages. (Without it, you would have to manually type each URL.)
* Import the Link component: “import { Link } from ‘react-router’
* Use the “to” property of the Link component to specify where the link should go to.
* The children of each Link component will be a hyperlink to the specified page.
* Use the “activeClassName” property of the Link component to specify the class that the Link’s child components should have when the link is selected.
* Example:

export const Menu = () =>

<nav className="menu">

<Link to="/" activeClassName="selected">

<HomeIcon />

</Link>

<Link to="/add-day" activeClassName="selected">

<AddDayIcon />

</Link>

<Link to="/list-days" activeClassName="selected">

<ListDaysIcon />

</Link>

</nav>

Using route parameters

* You can pass parameters into a component from the URL.
* Syntax: path = ‘/some/path/:parameter’
  + Example: “<Route path="list-days/:filter" component={App}>” will pass a parameter called filter. For examples, the URL “/list-days/backCountry” will pass in the value “backCountry” as the filter parameter. This can then be used as a props.
* The component can use the parameter. All parameters appear under “this.prop.params”
  + Example: The App component would then access the backCounty parameter as so: “this.prop.params.backCountry”

Nesting Routes

* You can have routes within routes (i.e. have route components that are children of route components), which allow you to modify certain parts of a webpage.
* Example:

<Route path="/" component={Left}>

<Route path="about" component={About} />

<Route path="members" component={MemberList} />

* + The homepage “/” renders the Left component, “/about” renders both the Left and the About component, and “/members” renders both the Left and the memberList component.

**Forms and Refs**

Creating a form component

* The “for” HTML label attribute is used to indicate which form component the label is labelling. However, since “for” is a reserved expression in JSX, it is replaced with “htmlFor”
* Good practice to use PropTypes for forms to ensure that the property values that the user gives are valid.

Using refs in class components

* Props are the only way parent components can interact with their children components.
  + Modify children by re-rendering the child with new props.
* Components can get a reference to JSX-created elements by adding a refs attribute to those elements. Parents can later access such elements.
  + Those components then refer to those elements by using “this.refs”.
* Whenever using ES6 class syntax, you must bind any custom-defined methods to the class in the constructor (refer to below example).
* Example:

export class AddDayForm extends Component {

constructor(props) {

super(props)

this.submit = this.submit.bind(this)

}

submit(e) {

e.preventDefault() // prevents the form from being cleared

console.log('resort', this.refs.resort.value)

}

render() {

const { resort, date, powder, backcountry } = this.props

return (

<form onSubmit={this.submit} className="add-day-form">

<label htmlFor="resort">Resort Name</label>

<input id="resort"

type="text"

required

defaultValue={resort}

ref="resort"/>

)

}

}

Using refs in stateless components

* Since stateless components aren’t classes and hence don’t have “this.refs”, use callback functions to get references.
* Example:

export const AddDayForm = ({ resort}) => {

let \_resort

const submit = (e) => {

e.preventDefault()

console.log('resort', \_resort.value)

}

return (

<form onSubmit={submit} className="add-day-form">

<label htmlFor="resort">Resort Name</label>

<input id="resort"

type="text"

required

defaultValue={resort}

ref={input => \_resort = input}/>

)

}

Two-way function binding

* Two-way function binding is how child components can pass information back up to their parents.
* The parent passes as a property to its child component a callback function. This callback function takes in a parameter with data that the child component receives and will update the parent state using that data. The child calls this callback function when it receives this updated data.
* Example:

//App class in App.js:

addDay(newDay) {

this.setState({

allSkiDays: [

//Spread operator. This populates the list with all entries

//that already exist in this.state.allSkiDays.

...this.state.allSkiDays,

newDay

]

})

}

render() {

…

AddDayForm onNewDay={this.addDay}/>

}

//AddDayForm.js:

export const AddDayForm = ({ resort, onNewDay }) => {

//This value is set when the form is submitted. Example code of how

//this is done is shown in previous examples.

let \_resort

const submit = (e) => {

e.preventDefault()

onNewDay({

resort: \_resort.value,

})

\_resort.value = ''

}

}

Adding an autocomplete component

* The datalist HTML5 tag can be used to autocomplete entries. (It only display entries that contain the text you have typed as you search within a list of items.)
* Use the map function() to dynamically populate all of the valid options, or manually type out each option.
* Example:

class Autocomplete extends Component {

const tahoeResorts = [ … ]

let \_resort

get value() {

return this.refs.inputResort.value

}

set value(inputValue) {

this.refs.inputResort.value = inputValue

}

render() {

return (

<div>

<input ref="inputResort"

type="text"

list="tahoe-resorts" />

<datalist id="tahoe-resorts">

{this.props.options.map(

(opt, i) =>

<option key={i}>{opt}</option>)}

</datalist>

</div>

)

}

return(

<Autocomplete options={tahoeResorts}

ref={input => \_resort = input}/>

)

}

**The Component Lifecycle**