CIT 382
Web Dev II

Week 3

Winter 2018

Phil Colbert

React

- React is an open-source JavaScript library for building user interfaces maintained by Facebook, Instagram and a number of other developers and corporations
- Meteor supports React, and we will be focusing this week on learning React as you work through the Meteor React simple-todos tutorial
- Just as with other packages, you will need to add React modules to your Meteor application
- React focuses on components defined within JSX files with a file extension of jsx

JSX Files

- JSX files contain XML-like syntax to be used by preprocessors to transform the JSX syntax into JavaScript (more specifically ECMAScript)
  - Preprocessors, also known as transpilers, parse through source code in one language and convert that code to code in another language
- React JSX component files and components start with an upper case letter, use the camel-case naming convention, and a lowercase .jsx file extension
  - As we'll see later lowercase and uppercase elements are treated differently by the JSX transpiler
- JSX files are NOT template files using a template language like Spacebars

React Packages

- React packages typically include the following
  - react: entry point React library
    - Component: base class for React components
  - react-dom: DOM specific React methods
- These packages are included using ES6 import statements
  - The import technique used to import react and react-dom depends on what methods or properties are needed

ES6 Import and Export

- The Meteor React tutorials utilizes a number of different ES6 import statement formats to load external JavaScript modules into the code
- Let's work through an online tutorial that discusses ES6 import and export statements
  - JavaScript Modules the ES6 Way
    - Exporting variables and functions
    - Importing variables and functions
    - Single default export for entire library import
      - Default exports unnamed
    - Curly brace usage in exports and imports
    - Mixing default export and named exports

Meteor React Tutorial Imports

- Let’s now examine some of the import statements in the Components step of the Meteor React simple-todos tutorial
  - Default named entire library imports

    ```
    import React from ‘react’
    import App from ‘../imports/ui/App.jsx’
    ```

  - Named imports

    ```
    import { Meteor } from ‘meteor/meteor’;
    import { render } from ‘react-dom’
    ```

  - Default and named imports (mixed)

    ```
    import React, { Component } from ‘react’
    ```

  - Note: If you look into the node_modules folder and look at the export techniques, you will see a mixture of older and new syntax.

Class .. Extends?

- ES6 formalizes a common syntax for creating classes using JavaScript
- You may have seen “function” classes, effectively creating a class constructor

```
function myClass(someVar) { }
```

- ES6 introduces the class keyword

```
class myClass {
      constructor (someVar) { }
}
```

- Using the ES6 extends keyword will create a new class that inherits all of the properties and methods of the parent class
- Note: Extending or inheriting from a class is also referred to as subclassing

Other ES6 Help

- Meteor, React and JSX use a number of ES6 features
- To learn more about some of these ES6 features, you will be working on the following pages during lab this week
  - Arrow functions
  - Classes
  - Template literals
  - let
  - const

React Component

- The App.jsx file (imports/ui) defines an exports a new class App that inherits (extends) from the React Component class

```
export default class App extends Component
```

  - All methods and properties of Component will be accessible from the new App class

- Remember the mixed import statement?

```
import React, { Component } from ‘react’
```

- We could have skipped the named Component import, and in the App class export statement simply included the entire library name and use dot notation to access Component

```
export default class App extends React.Component
```

Component render()

- When defining classes a class may establish a requirement that specific methods be defined by any class that extends the class
- The React Component class requires that the render() method be defined
- The render() method is used to return the display content of the new component using the return keyword
- We'll look more closely at the render() method as we analyze the App.jsx file

Back to JSX

- JSX syntax essentially translates into calls to create JavaScript elements using createElement (see JSX In Depth)
  - The online BABEL transpiler demonstrates JSX converted into JavaScript, and we'll use the online transpiler in explaining JSX
- We can also use a React CodePen sample to see React rendering in real time

JavaScript Array Map Refresher

- The sample Babel JSX contains the following code:

```
[1,2,3].map(n => n + 1);
```

- The JavaScript array map() method will call a function for every element in array
- The above sample also includes the ES6 standard arrow functions for defining a function
- ES6 arrow functions are defined by:
  - Function argument(s), arrow, function body
- Babel converts the above sample into the more familiar function format

```
"use strict";
[1, 2, 3].map(function (n) {
      return n + 1;
});
```

Sample React App

- A sample Meteor React app is available on Github based on the Facebook React tutorial that begins with Hello World
  - Github Hello World sample app (expanded)
- This sample app demonstrates how to work with Meteor, React and JSX components
- The sample app includes numerous comments explaining the code, tips and descriptions of techniques used
- Let's explore this sample app

Exports and Imports

- The sample app contains working code and comments regarding exports and imports
  - Exports
    - Default using class
    - Default using function
    - Named using class
    - Named using function
  - Imports
    - Default
    - Named
    - Mixed
  - Meteor packages
  - React packages
  - JSX components

Classes and Props

- The sample app includes information on using ES6 class syntax and React regarding
  - Constructors
  - Calling a base class
  - Props (read-only properties for transferring information to JSX component)
  - State (read-write class properties)
  - React Component class lifecycle methods
  - React Component render() method
  - Using React PropTypes to require component props
  - Class/component encapsulation of functionality

React JSX

- The sample app includes numerous examples and tips of how to use JSX and React components
  - A JSX component must return a single element only
  - A JSX component may return null if no component output is desired
  - JSX expressions and conditionals
  - Events and binding element *this* object
  - Forms and controlled components
- Tip: Use the HTML to JSX Compiler to convert normal HTML into JSX

Meteor Check and ES6

- The sample app includes using the meteor check package to validate data
- The sample app also includes examples and tips of using the ES6 arrow anonymous method syntax, including with no arguments, single arguments, and multiple arguments

Meteor React simple-todos App

- After analyzing the Hello World sample app, let's review the Meteor React simple-todos app and apply what we've learned
- Note: The Hello World is intended to assist you in understand the Meteor React, but the Meteor React tutorial includes other examples, tips and suggestions about how to layout Meteor React applications, and how to use other features of Meteor when developing React applications

Props and PropTypes

- React props represent data passed into a component as attribute values
- Props are read-only (immutable) within the component
- Props are referenced using this.props.property
- As component arguments, we can use PropTypes to enforce
  - Earlier versions of React included PropTypes as part of the React framework, but now PropTypes must be explicitly included
    - prop-types / Github prop-types
    - Use npm install --save prop-types to add to a project

Adding Props to a Component

- To add props to a component, you simply add attributes when declaring a component instance

```
import React, { Component } from 'react';
import MyComponent from './MyComponent';

export default class App extends Component {
  render() {
    return (
      <div>
        <MyComponent
          firstParam = "test"
          secondParam = {2+3}
        />
      </div>
    );
  }
}
```

Props as Attributes

- The sample code demonstrates that prop attribute values can be expressed as literals, or as JavaScript variables, expressions, or function calls by surrounding with curly braces
- Use meaningful names for props
- Props are accessible within a component using this.props.propertyName
- Props are also available as method parameters such as within the constructor

Props

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

export default class MyComponent extends Component {
    constructor(props) {
        super(props);
            console.log(props.firstParam);
    }
    render() {
        return (
            <div>
                {this.props.firstParam}<br />
                {this.props.secondParam}
            </div>
        );
    }
}
```

PropTypes

- Passing data to a component using props is easy, but error prone
- A component may require specific props to function correctly
- A component also should be able to expect props be of a specific data type
- A method should be available to establish default values for optional props
- Use PropTypes
  - prop-types / Github prop-types
  - Use npm install --save prop-types to add to a project

PropTypes

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

export default class MyComponent extends Component {
  render() {
    return (
      <div>
        {this.props.firstParamRequired}<br />
        {this.props.secondParamOptional}
      </div>
    );
  }
}
MyComponent.defaultProps = {
  secondParamOptional: false,
};
MyComponent.propTypes = {
  firstParamRequired: PropTypes.string.isRequired,
  secondParamOptional: PropTypes.bool,
};
```

React UI Errors

- As with most programming languages, JavaScript supports trapping errors using try/catch/finally
- Trapping React UI (user interface) errors requires a more sophisticated approach using an error component
  - React UI errors will produce cryptic errors, or even crash, if not handled
- React Error Boundaries provide a mechanism for creating a “wrapper” component for trapping Rect UI errors

Error Boundaries

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Display fallback UI
    this.setState({ hasError: true });
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Error Boundaries

- You will need to adapt the sample ErrorBoundary component to work within your code
- As a component, ErrorBoundary surrounds all output from your render() methods
- When an error occurs, the ErrorBoundary component will trap the error, and will output error information rather than your output

Error Boundaries

```
import React, { Component } from 'react';
import MyComponent from './ErrorBoundary';
import ErrorBoundary from './ErrorBoundary';

export default class App extends Component {
  render() {
    return (
      <div>
        <ErrorBoundary>
          <MyComponent />
        </ErrorBoundary>
      </div>
    );
  }
}
```

JSX

- React components require a render() method to return the reactive user-interface elements
- React render() statements are JSX (JavaScript Extensions)
- JSX elements are browser-independent DOM properties, attributes, and events
- JSX components are camel case (camelCase)
- JSX DOM Elements
- HTML to JSX Compiler

JSX Style Attributes

- As a general rule, you should use CSS to format and style your React components
- You may find a need to specify a style within your JSX code
- Style objects are declared within your Component, and use camelCase naming convention of typical CSS properties

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

render() {
  return <div style={divStyle}>Hello World!</div>;
}
```

React State

- React components include the concept of a state
- A React component’s state properties are read-write
- Changes to state will trigger a component’s render method
- Changes to state are made using setState(), except in a component constructor, where the state may be declared
- Changes to state also raise events within a component

State

```
export default class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      someStateString: "test",
      someStateBoolean: false,
      someStateObject: {aProperty: "Hi!"},
    };
  }
  changeState = () => {
    this.setState({someStateString: "changed"});
  }
  render() {
    return (
      <div>
        {this.state.someStateString}
      </div>
    );
  }
}
```

React Lifecycle Events

- Change to React components props or state will raise a number of React component lifecycle events
  - Note: If props are immutable, how can they change? Props are immutable within a component, but the parent component can always change a prop, causing the child component to receive new props, and thus raising prop and potentially state lifecycle events
  - Warning: Lifecycle events are very strict about whether or not the events can change the state. Changing the state within a state lifecycle event would essentially create an infinite loop, and typically will raise an error within React

Lifecycle

- Below are a few links to help you understand React lifecycle events:
  - React.Component
  - State and Lifecycle
  - Understanding the React Component Lifecycle

Clock Example

- Let’s create a React application to display a running date/time clock
  - This sample application is based on the clock application found at State and Lifecycle
- The app will have an App component, and a Clock component
- The Clock component will demonstrate the use of React lifecycles

Clock Example

- As experienced JavaScript programmers, we know that to display a continuously updated date/time, we'll need to use the Date object, and events
- Let's first look at a sample clock app using traditional JavaScript
  - Note: The traditional JavaScript version will use anonymous functions, but could have used ES6 arrow functions as well

Clock – Typical JavaScript

```
<!doctype html>
<html lang="en">
<head>
   <title>Clock JS</title>
   <meta charset="utf-8">
   <script>
window.onload = function() {
   setInterval( function() {
      document.querySelector('#dateTime').innerHTML
            = (new Date()).toLocaleString();
   });
};
   </script>
</head>
<body>
   <h1>Current date/time: <span id="dateTime"></span></h1>
</body>
</html>
```

Clock - React

- The React clock version definitely involves many more files
- Also, the React version involves using the basic React philosophy
  - A component has a state, and changes to state will be reflected in changes in the user interface
- We will also be using two React lifecycle events to enable and clear the timer
  - componentDidMount
  - componentWillUnmount

Clock - React

- Initially, let's create a basic Meteor React app (see Week 2 Lab instructions)
- The basic app needs to include a constructor, and within the constructor set a state variable to the current date
- Within the component render method, return the current date state variable

Clock - React

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

// App component - represents the whole app
export default class Clock extends Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date(),
    }
  }
  render () {
    return (
      <div>
        <h1>Current date/time: {this.state.date.toLocaleString()}</h1>
      </div>
    );
  }
}
```

Clock - React

- Now that we have the basics, we need to determine how to setup the timer, update the date state variable, and clear the interval event should the component be destroyed
- First, let's add the componentDidMount lifecycle event to setup the timer, and a callback function to update the date of the state variable

Clock - React

```
componentDidMount() {
    this.timerID = setInterval(
        () => this.tick(), 500
    );
}
tick () {
    this.setState({
        date: new Date(),
    });
}
```

Clock - React

- Finally, we need to clear the timer event
- Why? Clock is a re-usable component, and if we create and destroy the component multiple times, we need to remove the timers
- If we didn't remove the timers, it's possible that the browser is still generating the events, even if no one is listening

```
componentWillUnmount () {
  clearInterval(this.timerID);
}
```