LAB 20a

BEGINNING REACT

What You Will Learn

- How to use JSX to create components
- How to work with React collections such as props, state, and refs
- How to add behaviors to your React components

Approximate Time

The exercises in this lab should take approximately 75 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: Jan 16, 2019

LEARNING REACT

PREPARING DIRECTORIES

1. If you haven't done so already, create a folder in your personal drive for all the labs for this book.
2. From the main `labs` folder (either downloaded from the textbook's web site using the code provided with the textbook or in a common location provided by your instructor), copy the folder titled `lab20a` to your course folder created in step one.

This lab walks you through the creation of a few simple React applications. There are multiple ways of creating React. In this lab, you will begin with the simplest approach: using `<script>` tags to reference the React libraries and a `<script>` tag that will enable JSX conversion to occur at run-time. While this approach is certainly slower and not what you would do in a real-world application, it simplifies the process when first learning.

In the next React lab, you will take a better approach that puts each component in a separate file and which uses node, npm, and webpack to compile and bundle the application.

Exercise 20a.1 — USING JSX

1. Examine lab20a-ex00.html in the browser. We will be adding React functionality to this base page. It uses the Bulma CSS framework, which is a lightweight and clean framework. Why are we using it? No real reason, other than to try something new!
2. Copy the first `<article>` element to the clipboard (this will save you typing in step 4).
3. Open lab20a-ex01.html in a code editor.

   Notice that it already has the React JS files included via `<script>` tags. Notice also that it is using the Babel script library to convert our React JSX scripts at run-time.

4. Add the following JavaScript code to the head. Notice the `type="text/babel"` in the script tag. This is necessary because you will be entering JSX and not JavaScript in this tag.

   To save typing you can paste the `<article>` element you copied in step 2. Notice that you have to change the `class=` attributes to `className=` attributes. Remember, you are writing JSX and not JS.

```
<script type="text/babel">

/* There are several ways of creating a React component.
   The newer approach is shown here: using a class
*/
```

```
class Company extends React.Component {
  render() {
    return (
      <article className="box media ">
        <div className="media-left">
              <figure className="image is-128x128">
                <img src="images/AMZN.svg" >
              </figure>
        </div>
        <div className="media-content">
            <h2>Amazon</h2>
            <p><strong>Symbol:</strong> AMZN</p>
            <p><strong>Sector:</strong> Consumer Discretionary</p>
            <p><strong>HQ:</strong> Seattle, Washington</p>
        </div>
        <div className="media-right">
          <button className="button is-primary">Edit</button>
        </div>
      </article>
    );
  }
}
/*
   We now need to add the just-defined component to the browser DOM
*/
ReactDOM.render(<Company />,
   document.querySelector('#react-container'));

</script>
```

Note that JSX is class sensitive and follows XML rules.

5 Test in browser.

This code won't work.

6 Go to the JavaScript console and examine the error message.

You will see that it wanted the `<img>` tag to have a closing end tag. Why? JSX is an XML-based syntax (just like the old XHTML was), and thus your JSX must follow XML syntax rules: case sensitive, all tags must be closed, and all attributes in quotes.

7 Fix the code by adding a close tag to the `<img>` element:

```
<img src="images/AMZN.svg" />
```

8 Save and test in the browser. It should display a single `<Company>` element.

Copyright © 2020 Randy Connolly and Ricardo Hoar

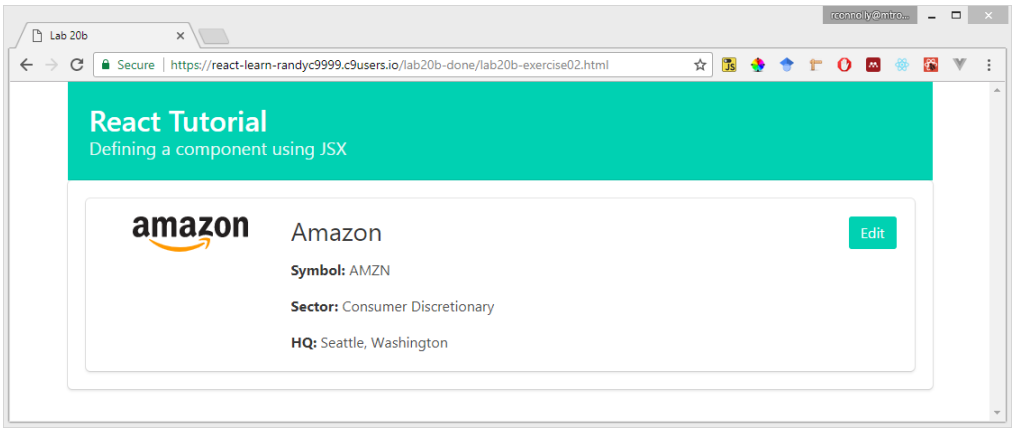

Figure 20a.1 – Finished Exercise 20a.01

Exercise 20a.2 — Adding additional components instances

1 Open lab20a-ex02.html.

In this exercise, you will add multiple instances of this component.

2 Add/modify the following code:

```
/* Notice here that we are defining another component which will
   contain other other components

   For this to work there must be a single root element being
   rendered ... try removing the <div> and it will no longer work
*/

const app = (
  <div>
      <Company  />
      <Company  />
      <Company  />
  </div>
);

ReactDOM.render(app, document.querySelector('#react-container'));
```

3 Test in browser.

The result should look similar to that shown in Figure 20a.2.

4 Comment out the code added in step 2 and replace it with the following:

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

5 Modify the `ReactDOM.render` call as follows and test. The results should look similar to that shown in Figure 20a.2.

```
ReactDOM.render(<App />, document.querySelector('#react-container'));
```

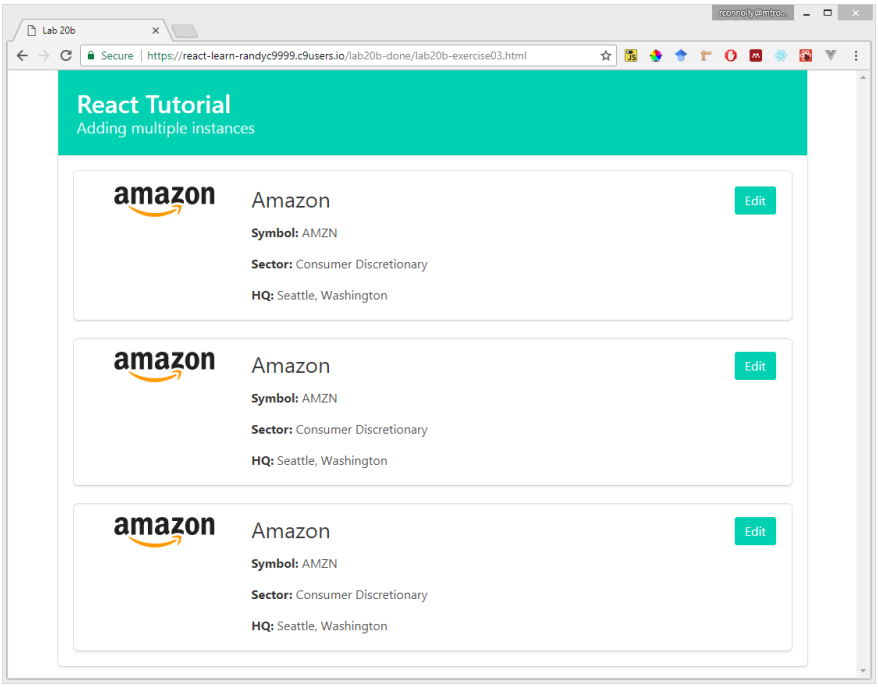

Figure 20a.2 – Finished Exercise 20a.02

6 In the example so far, React is only being used to implement the `App` component that consists only of the `Company` elements. It is much more common for React to be responsible for *all* the HTML elements in the document. Modify the App class by moving the rest of the markup from <main> element into the App element as follows:

```
class App extends React.Component {
  render() {
    return (
```

Copyright © 2020 Randy Connolly and Ricardo Hoar

```
        <main className="container">
            <section className="hero is-primary is-small">
              <div className="hero-body">
                <div className="container">
                  <h1 className="title">React Tutorial</h1>
                  <h2 className="subtitle">
                    Adding multiple instances
                  </h2>
                </div>
              </div>
            </section>

            <section  className="content box ">
              <Company  />
              <Company  />
              <Company  />
            </section>
        </main>
      );
    }
}
```

7 Add an empty div to the body as follows:

```
<body>
  <div id='react-container'></div>
</body>
```

8 Test. It should work and appear the same as Figure 20a.2.

Of course, this exercise was limited by the fact that it displays the same Company content. In the next exercise, you will use `props`, a read-only collection of property data that is populated via your JSX markup, to rectify this limitation.

Exercise 20a.3 — Using Props

1 Open lab20a-ex3.html.

2 Modify the App component as shown in the following code:

```
/* Notice here that we are adding attributes (and a child element)
   to our <Company> elements. These will be accessing within the
   component via the props collection. */
class App extends React.Component {
  render() {
    return (
      <div>
        <Company symbol="AMZN" sector="Consumer Discretionary"
           hq="Seattle, Washington">Amazon</Company>
        <Company symbol="GOOG" sector="Information Technology"
     hq="Mountain View, California">Alphabet Inc Class A</Company>
        <Company symbol="AAPL" sector="Information Technology"
           hq="Cupertino, California">Apple</Company>
        <Company symbol="T" sector="Telecommunications Services"
           hq="Dallas, Texas">AT&T</Company>
      </div>
    );
  }
}
```

The use of the child element for the name is not necessary; it is here just to show how to use it instead of attributes.

3 Modify the component as follows (some markup omitted) and test.

```
/* Here we are using the props collection to access the element
  Attributes */
class Company extends React.Component {
  render() {
    return (
      <article className="box media ">
        <div className="media-left">
              <figure className="image is-128x128">
                <img src={"images/" + this.props.symbol + ".svg"} />
              </figure>
        </div>
        <div className="media-content">
            <h2>{this.props.children}</h2>
            <p><strong>Symbol:</strong> {this.props.symbol}</p>
            <p><strong>Sector:</strong> {this.props.sector}</p>
            <p><strong>HQ:</strong> {this.props.hq}</p>
        </div>
        ...
      </article>
    );
  }
}
```

Copyright © 2020 Randy Connolly and Ricardo Hoar

4 Some components are quite simple and contain no behavior. Such components are known as **functional components**, and can be created in a simpler manner than the class-based approach used so far. To try this, add the following code (you can cut it from your `App` class) to your `<script>` element outside of the two classes defined already:

```
const Header = function(props) {
  return (
          <section className="hero is-primary is-small">
            <div className="hero-body">
              <div className="container">
                <h1 className="title">React Tutorial</h1>
                <h2 className="subtitle">
                  { props.subtitle }
                </h2>
              </div>
            </div>
          </section>
    );
};
```

5 Replace the header markup in the `App` class with this new `Header` element and test.

```
class App extends React.Component {
  render() {
    return (
      <main className="container">
          <Header subtitle="Using Props" />

          <section  className="content box">
            ...
        </section>
      </main>
    );
  }
}
```

A function becomes a valid React component simply by being passed a single props object.

6 Change this function to arrow syntax as follows (some markup omitted) and test.

```
const Header = props => {
  return (
          <section className="hero is-primary is-small">
            ...
          </section>
    );
};
```

In the next two exercises, you will add behaviors to the components and make use of the `state` collection to edit data within a component.

Exercise 20a.4 — Adding Behaviors

1 Open lab20a-ex04.html and add the following code:

```
/* You can add event handlers (or any helper functions) to any
   Component class. In this example, you will be also wiring a
   click event handler in the JSX ...notice the camel case  */
class Company extends React.Component {
  edit() {
      alert("now editing");
  }

  render() {
    return (
      <article className="box media ">
        <div className="media-left">
              <figure className="image is-128x128">
                <img src={"images/" + this.props.symbol + ".svg"} />
              </figure>
        </div>
        <div className="media-content">
            <h2>{this.props.children}</h2>
            <p><strong>Symbol:</strong> {this.props.symbol}</p>
            <p><strong>Sector:</strong> {this.props.sector}</p>
            <p><strong>HQ:</strong> {this.props.hq}</p>
        </div>
        <div className="media-right">
          <button className="button is-primary"
                  onClick={this.edit}>Edit</button>
        </div>
      </article>
    );
  }
}
```

2 Test in browser by clicking on any of the edit buttons.

This isn't all the impressive perhaps. In the next exercise, we will change the rendering of the component based on whether it is in edit mode.

Copyright © 2020 Randy Connolly and Ricardo Hoar

Exercise 20a.5 — Adding State

1 Open lab20a-exercise05.html and add the following code:

```
/* While helpful, props are read-only meaning that a component can't
 change them. Most components still need data it can manipulate and
 change. This is achieved via State ...
   In this example, our first State variable will be a flag
 indicating whether or not we are in edit mode  */
class Company extends React.Component {
  /* class constructor sets up initial state */
  constructor(props) {
      super(props);
      this.state = {editing: false};

      /* Unfortunately, "this" isn't bound to the correct context in
 React when using methods inside classes. The work around is to call
 bind or use arrow function syntax, which will bind "this"
 correctly.    */
      this.edit = this.edit.bind(this);
      this.save = this.save.bind(this);
  }

  /* Here we are defining some helper functions that change state
    ... notice the use of setState() ... it's a bit like PHP, which
   uses  the setcookie() function to change a cookie, but $_COOKIE
   to retrieve a cookie */
  edit() {
      this.setState({editing: true});
  }

  save() {
      this.setState({editing: false});
  }
```

2 Rename the `render` function to `renderNormal` (don't test it yet though).

3 Add the following functions:

```
renderEdit() {
  return (
    <article className="box media ">
      <div className="media-left">
        <figure className="image is-128x128">
           <img src={"images/" + this.props.symbol + ".svg"} />
        </figure>
      </div>
      <div className="media-content">
        <h2><input type="text" className="input"
               defaultValue={this.props.children} /></h2>
        <p><strong>Symbol:</strong>
        <input type="text" className="input" defaultValue=
            {this.props.symbol} /></p>
        <p><strong>Sector:</strong>
        <input type="text" className="input"
```

```
            defaultValue={this.props.sector} /></p>
          <p><strong>HQ:</strong> <input type="text" className="input"
           defaultValue={this.props.hq} /></p>
        </div>
        <div className="media-right">
           <button className="button is-info" onClick={this.save}>
           Save</button>
        </div>
      </article>
    );
}

/* we will render the component differently depending on our state
(whether use has clicked edit button) */
render() {
  if (this.state.editing)
     return this.renderEdit();
  else
     return this.renderNormal();
}
```

When setting state, the `setState()` function merges the provided state items with other items (so long as they have different names).

State updates actually happen asynchronously, so it is possible that a state update might not have occurred immediately after setting it.

4 Test in browser.

The Edit button should change the rendering of the component (see Figure 20a.3). Pressing the Save button will return the component to its normal view. However, you will notice that any changes you made via edit form haven't been preserved.

5 In step 1, there was some trickery around the use of the `bind()` function. Why was this necessary? In JavaScript, the meaning of `this` within a function is its run-time context (i.e., who called the function). But when functions get passed as objects to other functions, the run-time context can get lost, and in such a case, the meaning of `this` will fall back to default binding (global context). To deal with such an eventuality, you can explicitly call the `bind()` function to bind `this` to the function or class. This is essentially what is happening in the constructor in step 1.

An alternative is to use arrow syntax, since within an arrow function `this` is defined by its lexical scope (that is, `this` is equal to the scope it is defined within, not its run-time context as is the case with normal functions).

Comment out the two binding lines in the constructor.

```
constructor(props) {
    super(props);
    this.state = {editing: false};
    // this.edit = this.edit.bind(this);
    // this.save = this.save.bind(this);
}
```

Copyright © 2020 Randy Connolly and Ricardo Hoar

6 Change the method definitions in the class to arrow syntax and test.

```
edit = () => {
    this.setState({editing: true});
}

save = () => {
    this.setState({editing: false});
}
```

This should work in the same manner and display result in Figure 20b.3.

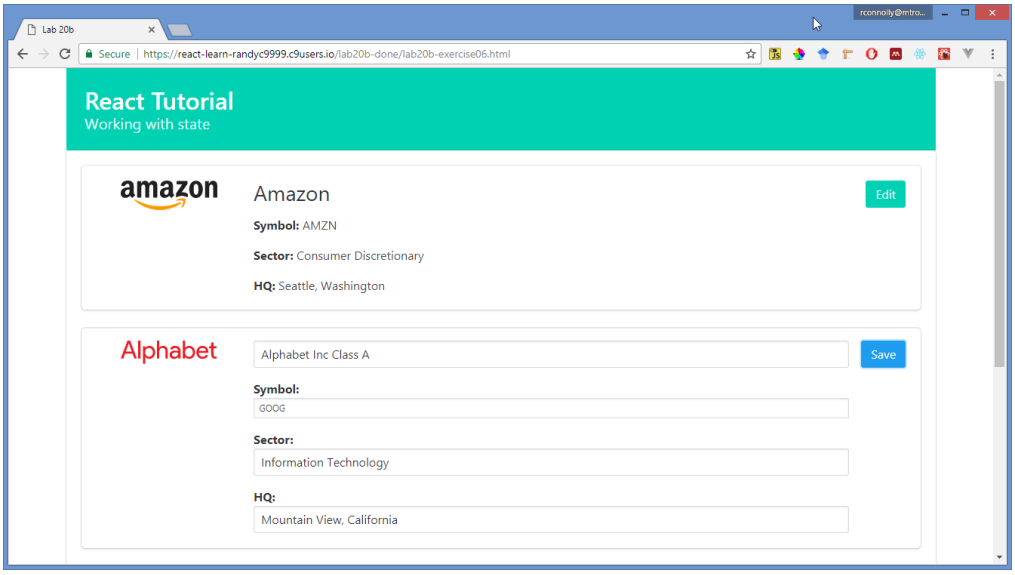

Figure 20b.3 – Finished Exercise 20a.05

Right now, the data for our components is provided by attributes when they are defined. It is more common, however, for a component's data to be provided dynamically at run-time, perhaps from some type of fetch call to a web API. To build towards that, the next exercise will add a new parent component that will be responsible for dynamically populating the individual `<Company>` elements (for now, just from an array).

Exercise 20a.6 — Creating a Parent Component

1 Open lab20a-ex06.html and add the following code:

```
/* In this example we are adding in a more sophisticated parent that
   contains Company data. Notice how we have moved the content out
   of the markup and into a data array in the component's state.
   Notice also then that the parent is responsible for populating
   the state of its children.  */

class Portfolio extends React.Component {

  /* the parent will contain the data needed for the children */
  constructor(props) {
     super(props);
     this.state = {
        companies: [
          {name: "Amazon", symbol: "AMZN", sector: "Consumer
Discretionary", hq: "Seattle, Washington"},
          {name: "Alphabet Inc Class A", symbol: "GOOG", sector:
"Information Technology", hq: "Mountain View, California"},
          {name: "Apple", symbol: "AAPL", sector: "Information
Technology", hq: "Cupertino, California"},
          {name: "AT&T", symbol: "T", sector: "Telecommunications
Services", hq: "Dallas, Texas"}
         ]};
  }
  /* This function will be responsible for generating a single
     populated Company element from the passed company data literal
  */
  createCompany(obj, ind) {
     return (<Company symbol={obj.symbol}
                      sector={obj.sector}
                      hq={obj.hq}
                      key={ind}
                      index={ind}>{obj.name}</Company>)
  }

  /* The render for this component will loop through our data and
     generate the appropriate Company elements */
  render() {
     return (
       <div> { this.state.companies.map(this.createCompany) } </div>
     );
  }
}
```

Copyright © 2020 Randy Connolly and Ricardo Hoar

2 Modify the App component as follows:

```
class App extends React.Component {
  render() {
    return (
      <main className="container">
          <Header subtitle="Creating Parent Component" />
          <Portfolio />
      </main>
    );
  }
}
```

3 Test in browser.

The result in the browser should be the same as the previous exercise.

Notice once again that the parent component was responsible for managing the data of the child (the `Company` elements). In the next exercise, you will add behaviors to this parent so that it manages the changes to the individual `Company` element's state. We will also add a Delete button to the edit state to demonstrate more functionality.

Exercise 20a.7 — Preserving State Changes

1 Open lab20a-ex07.html and modify the constructor method of the Company class (some code omitted):

```
constructor(props) {
    super(props);
    this.state = {editing: false};
    this.inputName = React.createRef();
    this.inputSymbol = React.createRef();
    this.inputSector = React.createRef();
    this.inputHQ = React.createRef();
}
```

2 Modify the `RenderEdit` method of the `Company` class (some code omitted):

```
renderEdit() {
  return (
    <article className="box media ">
      ...
      <div className="media-right">
        <button className="button is-info"
                onClick={this.save}>Save</button>
        <button className="button is-danger"
                onClick={this.delete} >Delete</button>
      </div>
    </article>
  );
}
```

3 In order to retrieve data from the DOM (e.g., the `<input>` element values), we need to first add the React `ref` attribute to those elements and references

```
renderEdit() {
  return (
    <article className="box media ">
      <div className="media-left">
            ...
      </div>
      <div className="media-content">
          <h2><input type="text" className="input"
             defaultValue={this.props.children}
             ref={this.inputName} /></h2>
          <p><strong>Symbol:</strong> <input type="text"
            className="input" defaultValue={this.props.symbol}
            ref={this.inputSymbol} /></p>
          <p><strong>Sector:</strong> <input type="text"
            className="input" defaultValue={this.props.sector}
            ref={this.inputSector} /></p>
          <p><strong>HQ:</strong>
          <input type="text" className="input"
            defaultValue={this.props.hq} ref={this.inputHQ} /></p>
      </div>
      <div className="media-right">
        <button className="button is-info"
                onClick={this.save}>Save</button>
        <button className="button is-danger"
                onClick={this.delete} >Delete</button>
      </div>
    </article>
  );
}
```

4 Add the following methods to the `Portfolio` class:

```
/* notice that the parent is responsible for making changes to the
   state  */
saveCompany = (newName, newSymbol, newSector, newHq, index) => {
    let tempArray = this.state.companies;
    /*
    remember that components change their state via setState()
    */
    tempArray[index] = { name: newName, symbol: newSymbol,
                         sector: newSector, hq: newHq };
    this.setState({companies: tempArray});
}

deleteCompany = (index) => {
    let tempArray = this.state.companies;
    tempArray.splice(index,1);
    this.setState({companies: tempArray});
}
```

Copyright © 2020 Randy Connolly and Ricardo Hoar

5 Since the `Company` components are the ones with the Save and Delete buttons, you will need to pass the handlers in the `Portfolio` to the `Company` components. To do so, modify the `createCompany` method in Portfolio as follows:

```
createCompany = (obj, ind) => {
   return (<Company symbol={obj.symbol}
                    sector={obj.sector}
                    hq={obj.hq}
                    key={ind}
                    index={ind}
                    saveData={this.saveCompany}
                    removeData={this.deleteCompany}>
           {obj.name}</Company>)
},
```

What's happening here? We are passing the save and delete methods defined in the parent to each child.

6 Now you will change the edit and delete event handlers for the buttons so they use the appropriate handlers in the parent. Add or modify the following event handlers in the `Company` class.

```
/* when we save, we're going to use refs to retrieve the user
   input and then ask the parent to save the data  */

save = () => {
    /* retrieve the user input */
    let newName = this.inputName.current.value;
    let newSymbol = this.inputSymbol.current.value;
    let newSector = this.inputSector.current.value;
    let newHq = this.inputHQ.current.value;

    /* via props we can call the functions in parent that have
       been passed to the child */
    this.props.saveData(newName, newSymbol, newSector, newHq,
                        this.props.index);
    this.setState({editing: false});
}

delete = () => {
    this.props.removeData(this.props.index);
    this.setState({editing: false});
}
```

7 Test in browser. You should be able to change and delete the data.

If you specify a symbol that exists in the image folder, the logo will change as well (or be displayed as a missing image)

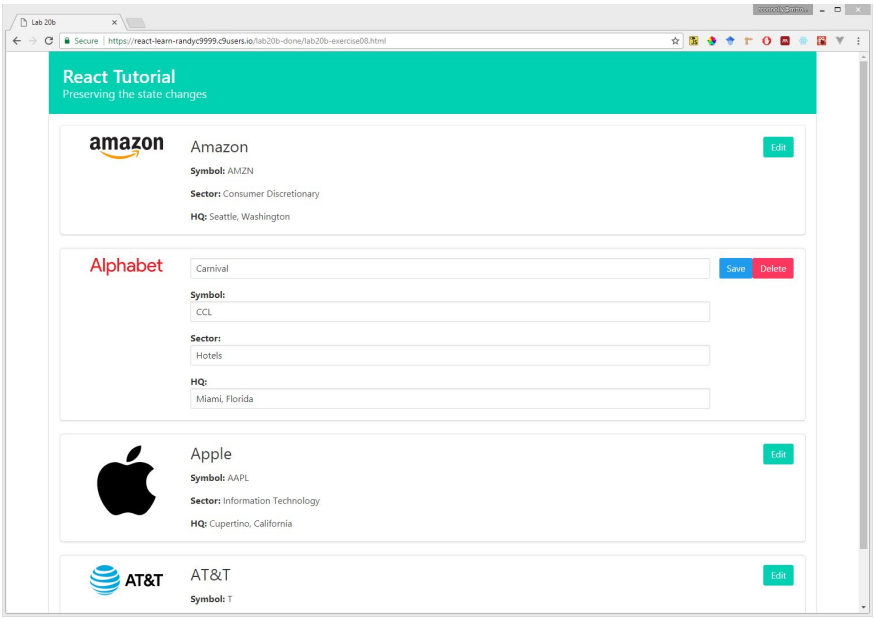

Figure 20a.4 – Finished Exercise 20a.07

Our last task will be to add a panel that allows to dynamically add `<Company>` elements based on user input.

Exercise 20a.8 — Adding Components to Parent

1 Open lab20a-exo8.html and add/modify the following code in `Portfolio` class:

```
addCompany = () => {
    let tempArray = this.state.companies;
    tempArray.push({ name: "New Company", symbol: "", sector: "",
                     hq: ""});
    this.setState({companies: tempArray});
}

render() {
    return (
        <div>
           <div className="box">
              <button className="button is-primary"
                      onClick={this.addCompany}>
               Add Company</button>
           </div>
           { this.state.companies.map(this.createCompany) }
        </div>
        );
}
```

2 Test in browser.

Copyright © 2020 Randy Connolly and Ricardo Hoar

Now it is your turn. The next exercise shows you what to create, but leaves it to you to implement it in React using the knowledge gained in the previous exercises.

Test Your Knowledge #1

1. Open lab20a-test01-markup-only.html in the browser.

   This provides the markup needed for the exercise. You will be implementing a reddit-style vote up/down component to each company. The number of votes will be shown in normal mode (see Figure 20b.5), but when in edit mode, you will display the up and down buttons with the current vote count displayed between the buttons (see Figure 20b.6). This markup file shows both so you know which markup to use.

2. You will be starting with lab20a-test01.html (it's the same as the finished previous exercise).
3. Modify the `companies` array by adding a `vote` property to each company object.
4. Modify `saveCompany`, `createCompany`, and `addCompany` functions in `Portfolio` component to include vote information (see additional steps below for guidance).
5. Modify the `renderNormal` function in `Company` component to display the number of votes (see Figure 20b.5).
6. Create a new React component that will implement the functionality for voting up/down (see Figure 20b.6). This component will store a single vote number in its state, and will be passed the initial value via props. The markup for element can be found in the markup file mentioned in step 1. The up and down buttons will need event handlers in the class that will increment the `vote` value stored in the state.
7. Modify the `renderEdit` function in `Company` component to display your vote component (see Figure 20b.5). This is simply a matter of adding a tag with the component name and passing, via an attribute, the correct vote data from the `companies` array. You will need to be able to access this element later, so be sure to give it a `ref` attribute.
8. You will need to modify the `save` function in the `Company` component to pass the current vote value to the `saveData` function. You can retrieve the vote value from the component by using its `ref` name (plus `.current`), the `state` property, and whatever its state variable name is in your vote component.
9. Finally, modify the `saveCompany` function in the `Portfolio` component to sort the companies array by the vote count (i.e., the company with highest number of votes is shown first). This is actually quite easy, though you will have to discover on your own how to sort an array of objects on a property's values.

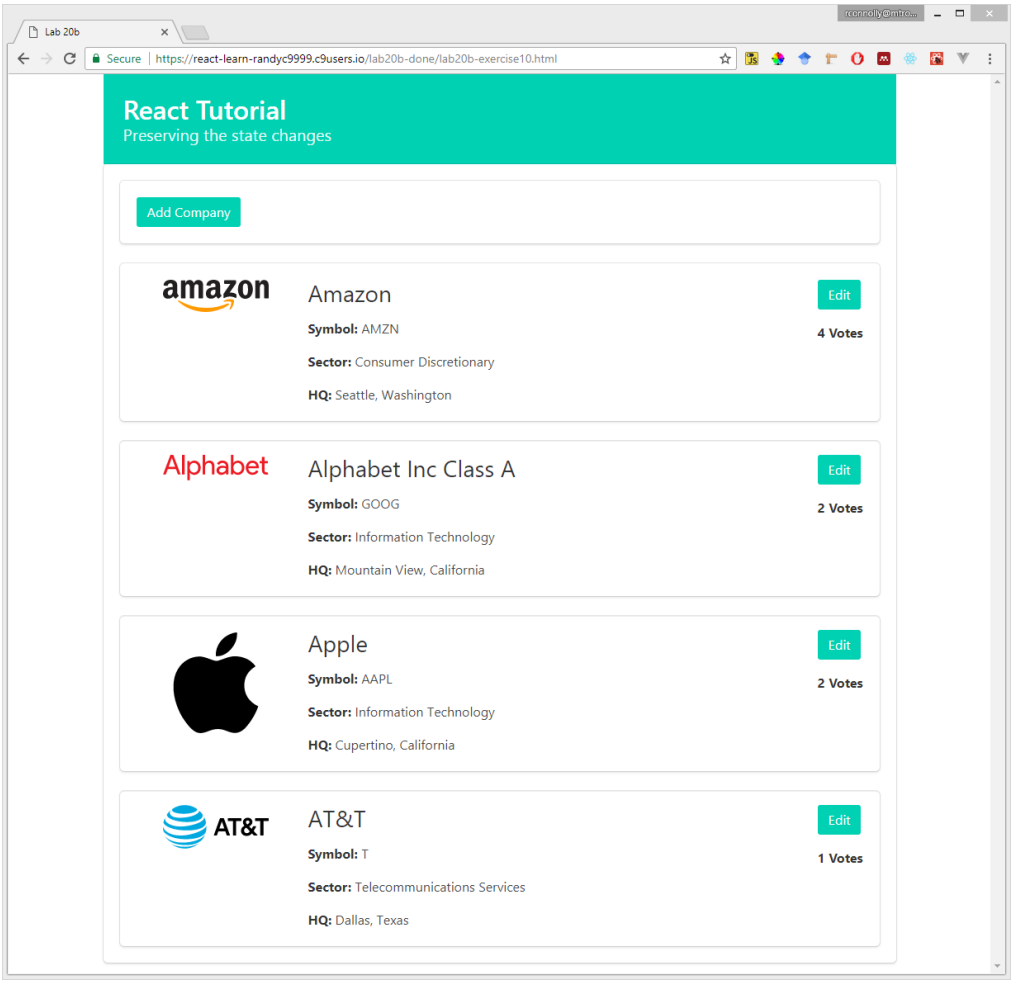

Figure 20a.5 –Exercise 20a.09 in normal mode

Copyright © 2020 Randy Connolly and Ricardo Hoar

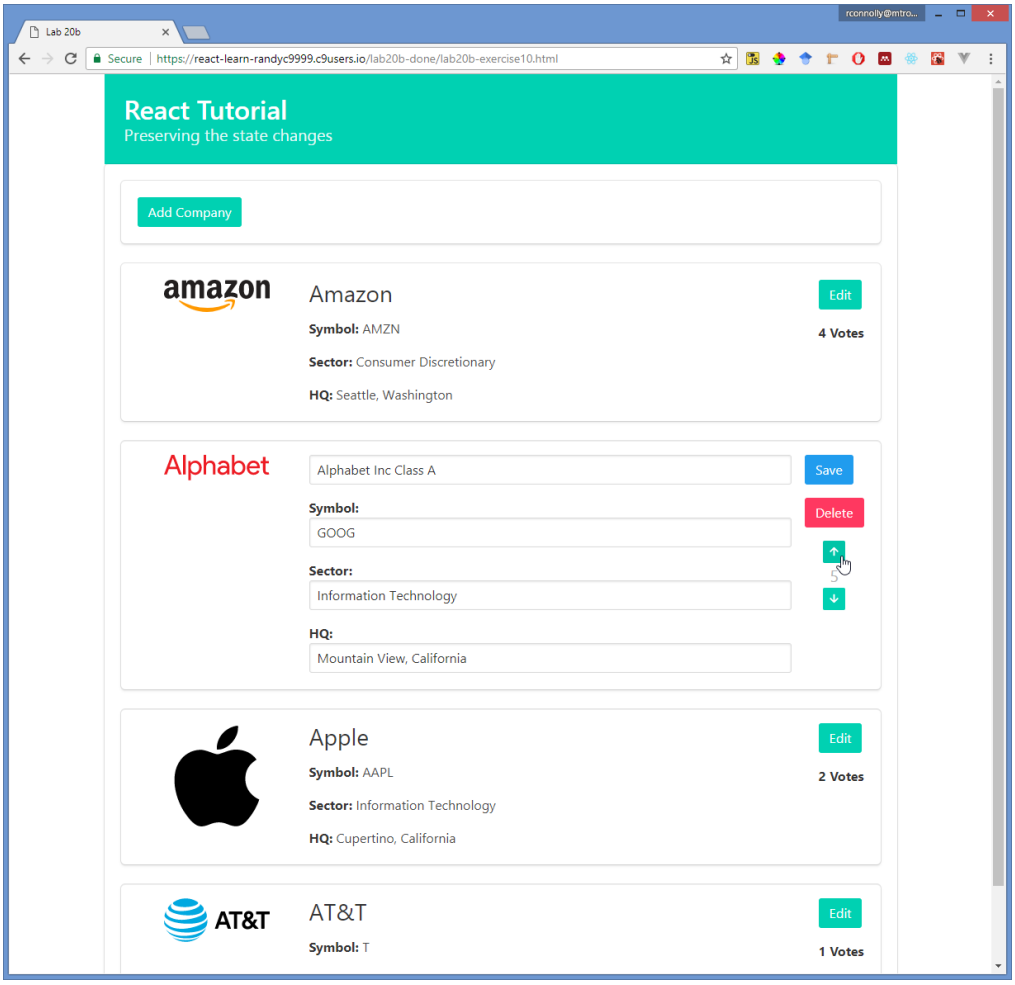

Figure 20a.6 –Exercise 20a.10 in edit mode