# 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 90 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 30, 2020

# 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 create-react-app along with node, npm, and webpack to compile and bundle the application.

## Exercise 20a.1 — USING JSX

1. Examine lab20a-ex00.html in the browser. This file partially illustrates some of the layout we will be implementing in React (this file is just simple HTML). 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. We will be adding React functionality to this base page.

   *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. This is fine for learning and simple examples (such as this lab), but isn't a suitable approach for a production site. Later, we will make use of CLI tools that convert the JSX into JavaScript.*

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.

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

/* There are several ways of creating a React component.
   Here we are using a ES6 class
*/
```

```
class Company extends React.Component {
  // notice we are using new ES7 property method shorthand syntax
  render() {
    // this is not JS but JSX
    return (
      <figure className="image is-128x128">
          <img src="images/FB.svg" >
      </figure>
    );
  }
}
/*
   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/FB.svg" />
```

8 Save and test in the browser. It should display a single `<Company>` element (which is for now simply an image).

9 Copy the `<figure>` element and paste it after the existing `<figure>`. Thus the component will try to return two `<figure>` elements.

10 Save and test in the browser.

*It won't work.*

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

*A rendered React component must have a single root element. Right now it has two <figure> elements so it won't work.*

12 Modify the component by adding the following code and test.

```
class Company extends React.Component {
  render() {
    return (
      <article className="box media ">
        <div className="media-left">
          <figure className="image is-128x128">
            <img src="images/FB.svg" />
          </figure>
        </div>
        <div className="media-content">
          <h2>Facebook</h2>
          <p><strong>Symbol:</strong> FB</p>
          <p><strong>Sector:</strong> Internet Software and
                    Services</p>
          <p><strong>HQ:</strong> Menlo Park, California</p>
        </div>
        <div className="media-right">
          <button className="button is-link">Edit</button>
        </div>
      </article>
    );
  }
}
```

*The result should look similar to that shown in Figure 20a.1.*

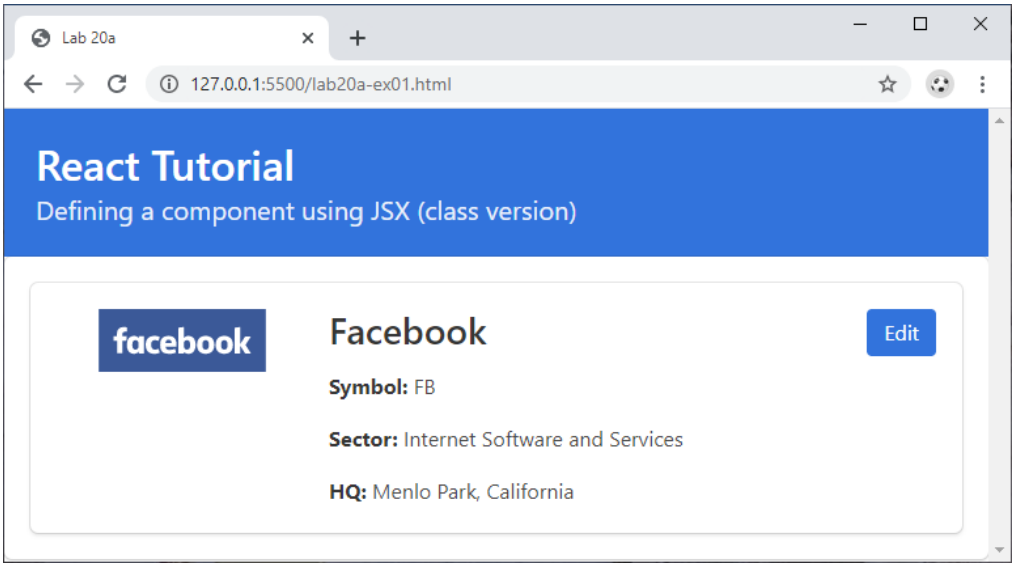


*Figure 20a.1 – Finished Exercise 20a.01*

## Exercise 20a.2 — Simple Functional Components

1 Open lab20a-ex02.html.

*In this exercise, you will encounter another way to create React components.*

2 Add the following code after the `Company` class:

```
/* This is another way to define a component: */
const simple = <h2>This is a component</h2>;
```

*This uses JSX to define a simple component.*

3 Modify the `ReactDOM.render` call as follows:

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

4 Test in browser.

*The result should display the new component.*

5 Comment out the code from step 2 and add the following new component.

```
const Simple = function() {
    return <h2>This is a functional component</h2>;
}
```

*This is a more common way to define a lightweight React component known as a functional component.*

6 Modify the `ReactDOM.render` call as follows:

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

*Note that functional components can be referenced as markup. Also, React expects functional components to begin with a capital letter.*

7 Modify your component as follows and test.

```
const Simple = function() {
    return <h2 className="is-size-1">
              This is a functional component</h2>;
}
```

*JSX can span multiple lines. Also, we can't use the HTML `class` attribute, but must use `className` instead.*

8 Comment out the code from step 2 and add the following new component.

```
const alternate = React.createElement(
  'h2',
  {className: 'is-size-1'},
  'This is an alternate version of the previous component'
);
```

9 Modify the `ReactDOM.render` call as follows:

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

*The Babel transcompiler (which converts from JSX to vanilla JavaScript) will convert the code from steps 6 and 7 into something similar to that shown in steps 8 and 9.*

## Exercise 20a.3 — Passing Parameters to a Component

1 Open lab20a-ex03.html and modify the component as follows.

```
const Simple = function(props) {
    return <h2 className="is-size-1">{props.title}</h2>;
}
```

2 Modify the `ReactDOM.render` call as follows and test.

```
ReactDOM.render(<Simple title="Using Props" />,
      document.querySelector('#react-container'));
```

*React passes the component's attributes as an object to the components. While we could have used any parameter name, later you will learn when we go back to using class-based components that this variable containing the passed attribute values is always called props.*

3 Modify the component and render call as follows and test.

```
const Simple = function(props) {
    return <h2 className="is-size-{props.size}">{props.title}</h2>;
}
ReactDOM.render(<Simple title="Using Props" size="3" />,
      document.querySelector('#react-container'));
```

4 Modify the component as follows (all on one line in your editor) and test.

```
const Simple = props => <h2 className="is-size-{props.size}">
                                    {props.title}</h2>;
```

*This is just a reminder that we can use arrow syntax in React.*

## Exercise 20a.4 — Adding Additional Component Instances

1 Open lab20a-ex04.html.

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

2 Add the following code after the `Company` class definition.

```
const app = <div>
               <Company />
               <Company />
               <Company />
            </div>;
```

3 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'));
```

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.

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

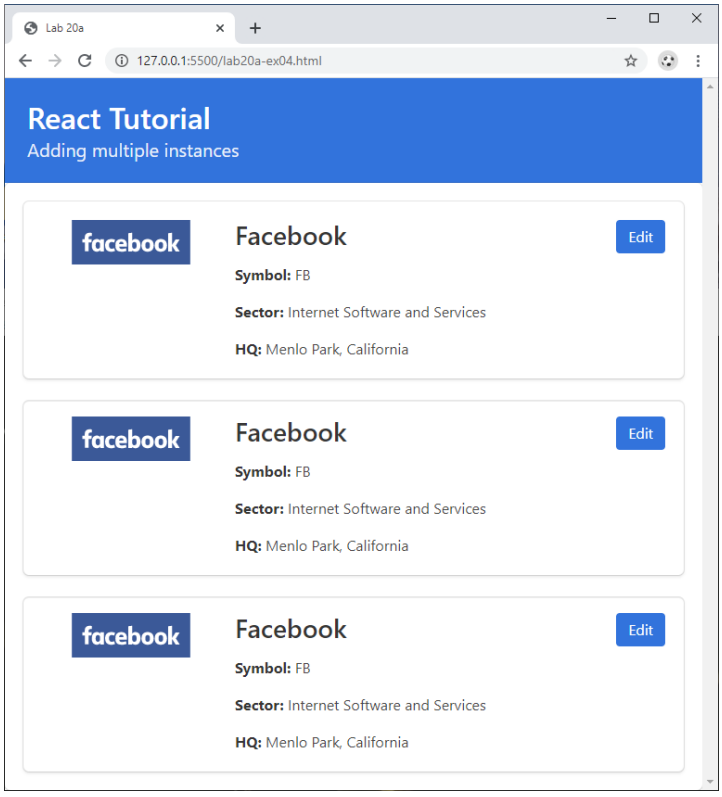


*Figure 20a.2 – Finished Exercise 20a.04*

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 (
      <main className="container">
          <section className="hero is-primary is-small
                                  has-background-link">
            <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.5 — Using Props

1 Open lab20a-ex05.html.

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

```
class App extends React.Component {
  render() {
    return (
      <div>
        <Company symbol="FB" sector="Internet Software"
           hq="Menlo Park, California">FaceBoook</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.

```
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>
    );
  }
}
```

4 Recall from an earlier exercise that **functional components** can be created in a simpler manner than the class-based approach. 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
                                has-background-link">
            <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>
    );
  }
}
```

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

```
const Header = props =>
          <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.6 — Adding Behaviors

1 Open lab20a-ex06.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-link"
                  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.*

## Exercise 20a.7 — Adding State

1 Open lab20a-exercise07.html and add the following code:

```
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 you will define helper functions that change state */
  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>
    );
  }

  /* render the component differently depending on our state (whether
  user 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);
}
```

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.*

7 Comment out the constructor you created in steps 1 and 5. Add the following code instead:

```
class Company extends React.Component {
    state = {
      editing: false
    };
    //constructor(props) {
    ...
```

*This is a second way to initialize state directly. The advantage of this approach is that it doesn't require the constructor boilerplate code.*

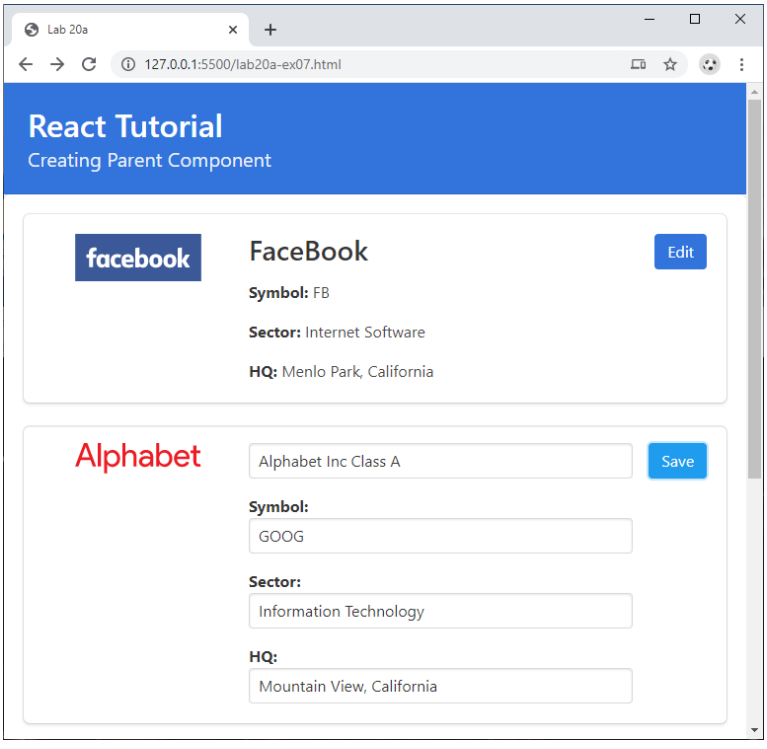


*Figure 20b.3 – Finished Exercise 20a.07*

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.8 — Creating a Parent Component

1 Open lab20a-ex08.html and add the following code:

```
class Portfolio extends React.Component {

  /* the parent will contain the data needed for the children */
  constructor(props) {
     super(props);
     this.state = {
        companies: [
          {name: "FaceBook", symbol: "FB", sector: "Internet
Software", hq: "Menlo Park, California"},
          {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, index) {
     return (<Company symbol={obj.symbol}
                      sector={obj.sector}
                      hq={obj.hq}
                      key={index}
                      index={index}>{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>
     );
  }
}
```

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.*

4 Replace the `createCompany` and the `render` methods with the following (it functions exactly the same but illustrates a different syntax).

```
render() {
   return (
     <div>
       { this.state.companies.map( (c, index) =>
            <Company symbol={c.symbol} sector={c.sector}
                     hq={c.hq} key={index}
                     index={index}>{c.name}</Company>
          ) }
     </div>
   );
}
```

Notice once again that the parent component was responsible for managing the data of the child (the `Company` elements). In a future exercise, you will add behaviors to this parent so that it manages the changes to the individual `Company` element's state. But before we get there, let's do a digression on Controlled versus Uncontrolled Components within Forms.

# FORMS IN REACT

Forms can operate differently in React, since form elements in HTML manage their own internal mutable state. For instance, a `<textarea>` element has a `value` property, while a `<select>` element has a `selectedIndex` property. With React, we can let the HTML form elements continue to maintain responsibility for their state (known as **uncontrolled components**), or we can let the React components containing the form elements maintain the mutable state (these are known as **controlled components**).

## Exercise 20a.9 — Controlled Form Components

1 Examine lab20a-ex09.html in the browser, and then modify the constructor method of the `SampleForm` class:

```
constructor(props) {
   super(props);
   this.state = {
     company: {
         name: "FunWebDev Corp",
         sector: "Textbooks",
         comments: "They know things!"
     }
   };
}
```

2 Add the following method to the `SampleForm` class:

```
handleSubmit = (e) => {
    e.preventDefault();
    let values = `Current values are
            ${this.state.company.name}
            ${this.state.company.sector}
            ${this.state.company.comments}`;
    alert(values);
}
```

3 Add a reference to this new method to the form, as follows:

```
<form className="container" onSubmit={this.handleSubmit} >
```

4 Test in browser. Modify the form data then click Submit.

*The form data isn't currently reflected in the state.*

5 Add the following methods to the `SampleForm` class:

```
handleCompanyChange = () => {
    const updatedCompany = { ...this.state.company };
    updatedCompany.name = event.target.value;
    this.setState({company: updatedCompany});
}
handleSectorChange = () => {
    const updatedCompany = { ...this.state.company };
    updatedCompany.sector = event.target.value;
    this.setState({company: updatedCompany});
}
handleCommentsChange = () => {
    const updatedCompany = { ...this.state.company };
    updatedCompany.comments = event.target.value;
    this.setState({company: updatedCompany});
}
```

*The spread operator (...) is used to make a copy of the contents of the company object that is already in state.*

6 Modify the forms element as follows (some markup omitted):

```
<input className="input" type="text"
        value={this.state.company.name}
        onChange={this.handleCompanyChange}  />

<select value={this.state.company.sector}
        onChange={this.handleSectorChange} >

<textarea className="textarea"
        value={this.state.company.comments}
        onChange={this.handleCommentsChange} ></textarea>
```

7 Test in browser. Modify the form data then click Submit.

*The alert data should now reflect the current form data. However, having separate change event handlers for each form element is unwieldy. The next steps provide a solution.*

8 Modify the forms element as follows (some markup omitted):

```
<input className="input" type="text" name="name"
       value={this.state.company.name}
       onChange={this.handleChange}  />

<select value={this.state.company.sector} name="sector"
       onChange={this.handleChange}  />

<textarea className="textarea" name="comments"
          value={this.state.company.comments}
          onChange={this.handleChange} ></textarea>
```

9 Comment out the event handlers added in step 5, add the following, and test:

```
handleChange = (e) => {
    const updatedCompany = { ...this.state.company };
    // this uses bracket notation to change property
    updatedCompany[e.currentTarget.name] = e.currentTarget.value;
    this.setState({ company: updatedCompany});
}
```

*This should work.*

10 We can make our code a little less verbose by using object destructuring. Add the following line to the `render()` method.

```
render() {
    const { name, sector, comments } = this.state.company;
    return (
```

11 Modify the forms element as follows (some markup omitted) and test.

```
<input className="input" type="text" name="name"
       value={name} onChange={this.handleChange}  />
<select value={sector} name="sector"
       onChange={this.handleChange}  />
<textarea className="textarea" name="comments" value={comments}
          onChange={this.handleChange} ></textarea>
```

While controlled components are generally recommended, for a complex form with many fields, validation, and style changes, it can get pretty tedious to use controlled components. An alternative is to use uncontrolled components, in which the state for each form element is managed by the DOM. And instead of writing event handler for every form element state update, you use **refs**.

### Exercise 20a.10 — Uncontrolled Form Components

1 Examine lab20a-ex10.html in the browser, and then modify the constructor method of the `SampleForm` class:

```
constructor(props) {
   super(props);
    // define some class properties for our element state
    this.name = React.createRef();
    this.sector = React.createRef();
    this.comments = React.createRef();
}
```

2 Modify the forms element as follows (some markup omitted):

```
<input className="input" type="text" ref={this.name}    />

<select ref={this.sector}>

<textarea className="textarea" ref={this.comments}></textarea>
```

3 Modify the handleSubmit method as follows:

```
handleSubmit = (e) => {
    e.preventDefault();
    let values = `Current values are
            ${this.name.current.value}
            ${this.sector.current.value}
            ${this.comments.current.value}`;
    alert(values);
}
```

4 Test in browser. Modify the form data then click Submit.

*The alert data should now reflect the current form data.*

Now, let's get back to our original example, where we will use uncontrolled form components.

## Exercise 20a.11 — Preserving State Changes

1 Open lab20a-ex11.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 of its children  */
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});
}
```

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) => <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 if you specify a symbol that doesn't exist in folder).*

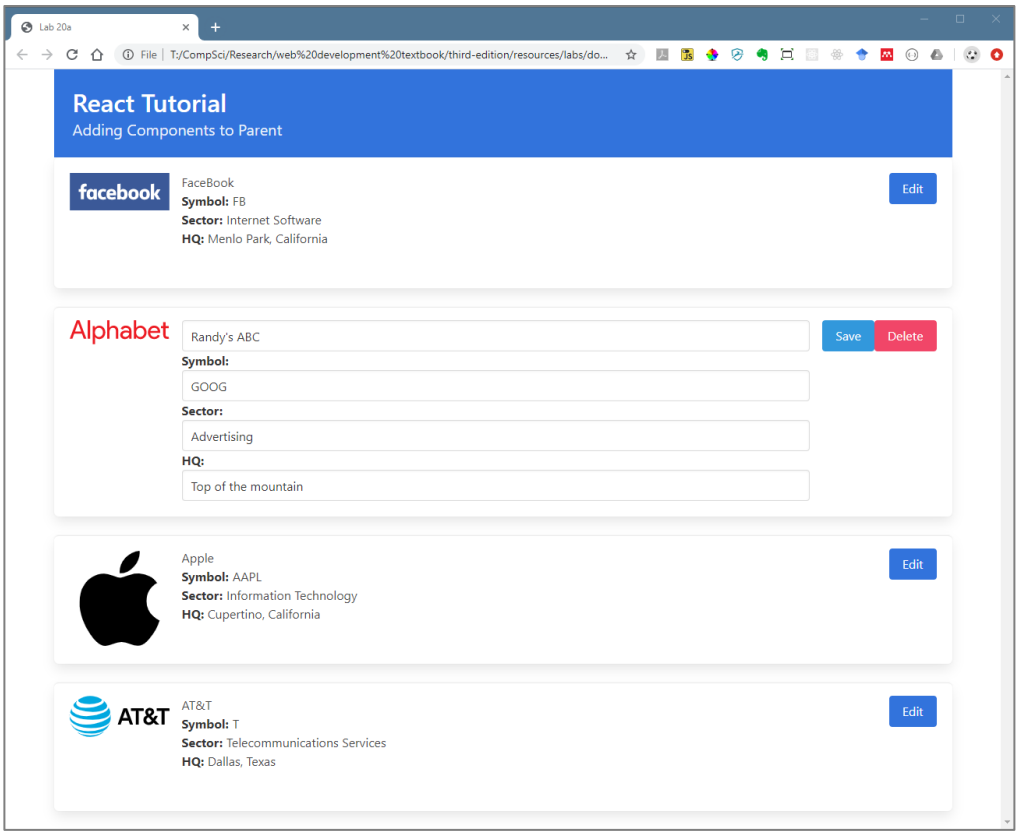


*Figure 20a.4 – Finished Exercise 20a.11*

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

## Exercise 20a.12 — Adding Components to Parent

1 Open lab20a-ex10.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-link"
                      onClick={this.addCompany}>
               Add Company</button>
           </div>
           { this.state.companies.map(this.createCompany) }
        </div>
        );
}
```

2 Test in browser.

Now it is your turn.

## Test Your Knowledge #1

1. Figure 20a.5 illustrates what the finished version should look like and the provided component hierarchy. Changing any of the form fields on the right will update the movie display on the left. The starting of the components and their `render()` methods have been provided. The data is contained within the file movie-data.js.

2. Implement the rest of the `MovieList`, `SingleMovie`, and `MovieLink` components.

   For `MovieList`, you will need to render a `<SingleMovie>` for each movie in the passed list of movies using `map()`. You will need to pass a movie object to `SingleMovie`.

   For `SingleMovie`, you will need to replace the sample data with data from the passed-in movie object. In the footer area, you will render a `<MovieLink>` and pass it the `tmdbID` property from the movie object.

   `MovieLink` must be a functional component. It will return markup similar to the following:

```
<a className="button card-footer-item"
   href="https://www.themoviedb.org/movie/1366" >
        <img src="images/tmdb.svg" width="30" />
</a>
```

   You will replace `1366` with the passed `tmdbID` value.

3. In the `App` component, you will add the `<MovieList>` component to the render. Be sure to pass it the list of movies in state. Test.

4. In the `App` component, use `map()` to output a `<MovieForm>` for each movie. Be sure to pass both `index` and `key` values to each `MovieForm`. Also pass the `saveChanges` method to each `MovieForm`. Test.

5. Make `MovieForm` a Controlled Form Component. Just like Exercise 20a-9, this will require creating some type of handler method within `MovieForm` that will call the `saveChanges` method that has been passed in (see also next step).

6. Implement `saveChanges` in the `App` component. Notice that it expects a movie object that contains within it the new data. Your method will use the index to replace the movie object from the movies data with the new data, and then update the state. Test.

   *It should work and look like Figure 20a.5. Making any changes to any form field should be reflected in the MovieList.*

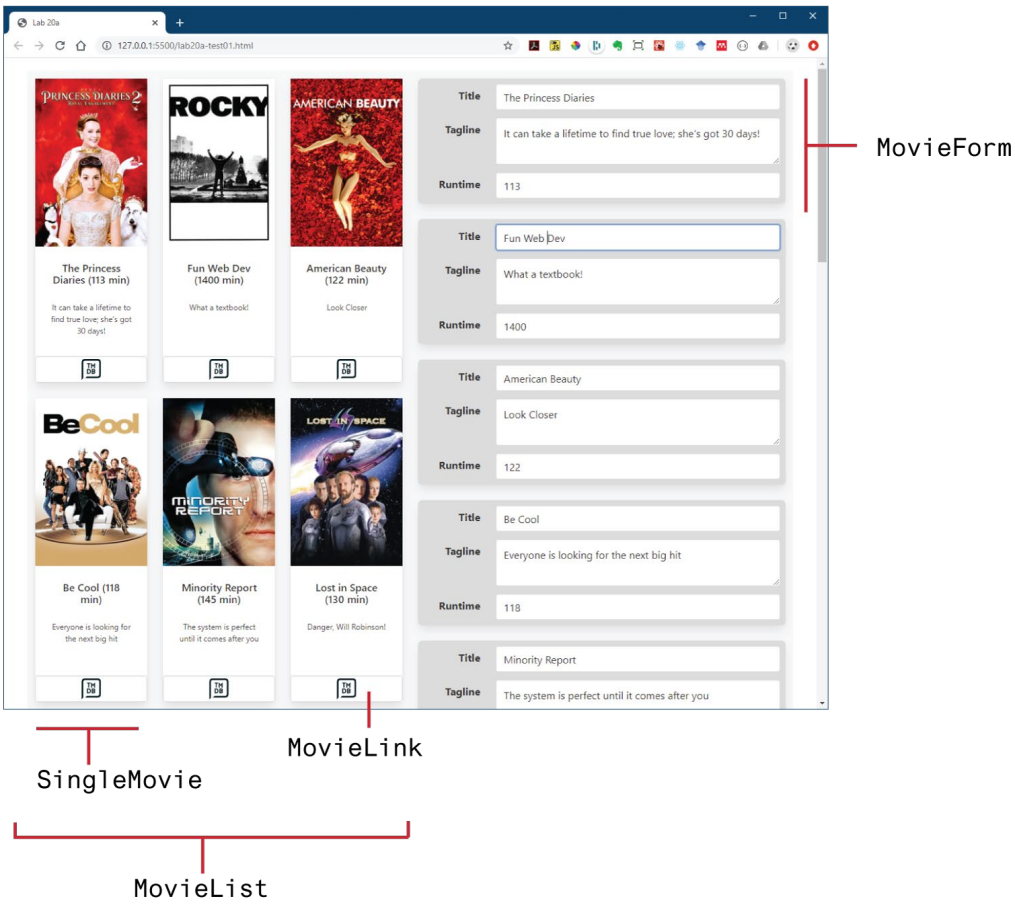


*Figure 20a.5 – Finished Test Your Knowledge #1*

## TEST YOUR KNOWLEDGE #2

1. Open lab20a-test02-markup-only.html in the browser.

   *This provides the markup needed for the exercise.*

2. You will be starting with lab20a-test02.html. Notice that it contains just a single container tag in the <body>. You will generate the rest of the markup using React code to be contained in lab20a-test02.js.

   *Using an external script with React will require an HTTP server (otherwise it generates a CORS error). If you are using an editor such as Visual Code, you can make use of an extension such as Live Server. Alternately, you can put your React code within a <script> tag inside of lab20a-test02.html.*

3. The data array is already defined and included in the file movie-data.js. Your code will eventually display each movie in the array as a `<li>` item. Clicking a movie's heart `<button>` will add the movie to the favorites list

4. Create the React components as shown in Figure 20a.6. The markup-only file had no behaviors: here you will implement the add-to-favorites functionality via React. The result should look similar to Figure 20a.7.

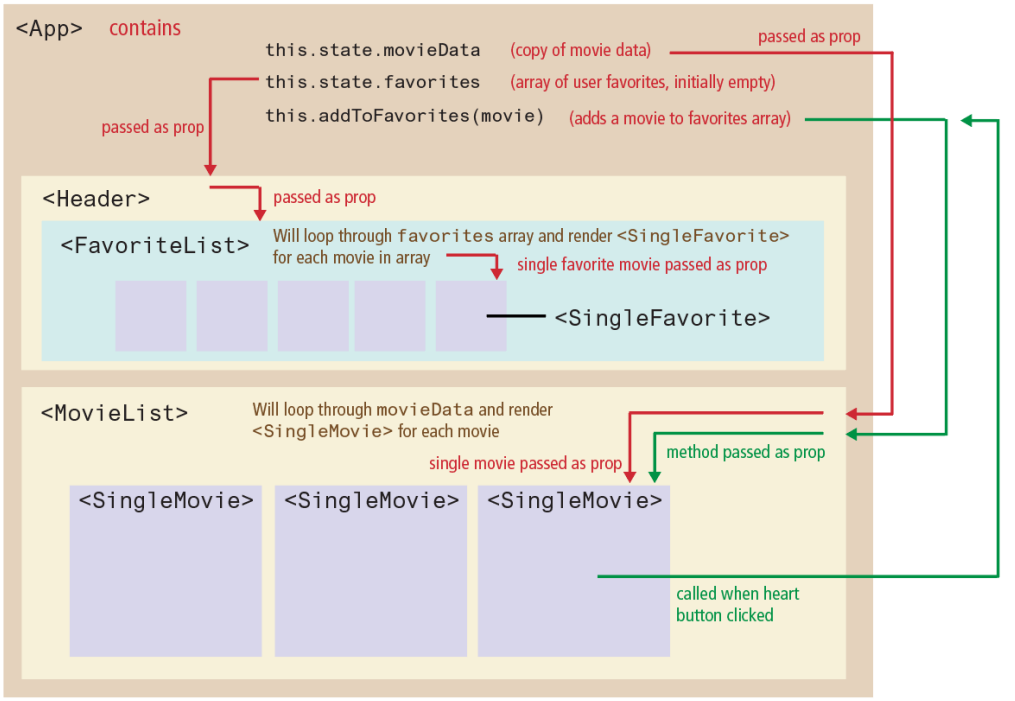


*Figure 20a.6 – Data flow between components*

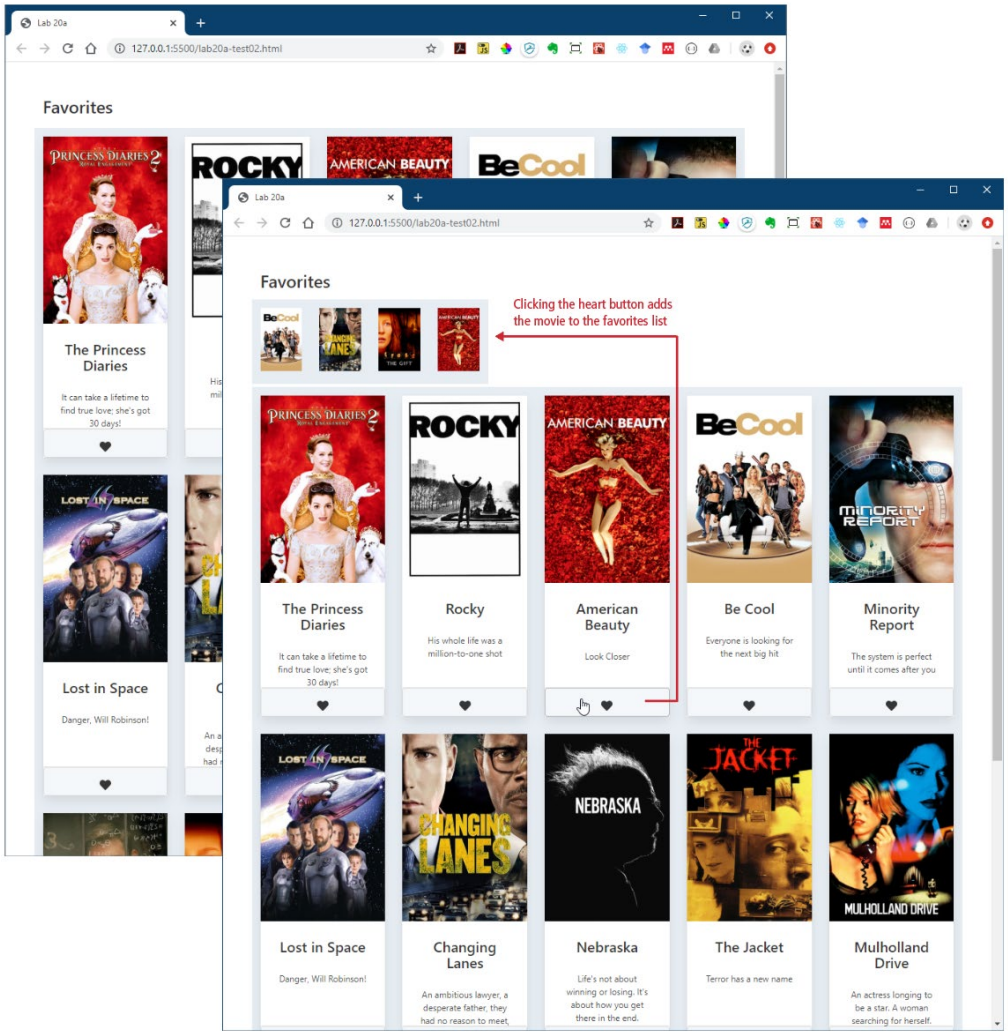


*Figure 20a.7 – Final Result*

## Test Your Knowledge #3

1. Open lab20a-test03-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.8), 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.9). This markup file shows both so you know which markup to use.*

2. You will be starting with lab20a-test03.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.8).
6. Create a new React component that will implement the functionality for voting up/down (see Figure 20b.9). 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.9). 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.

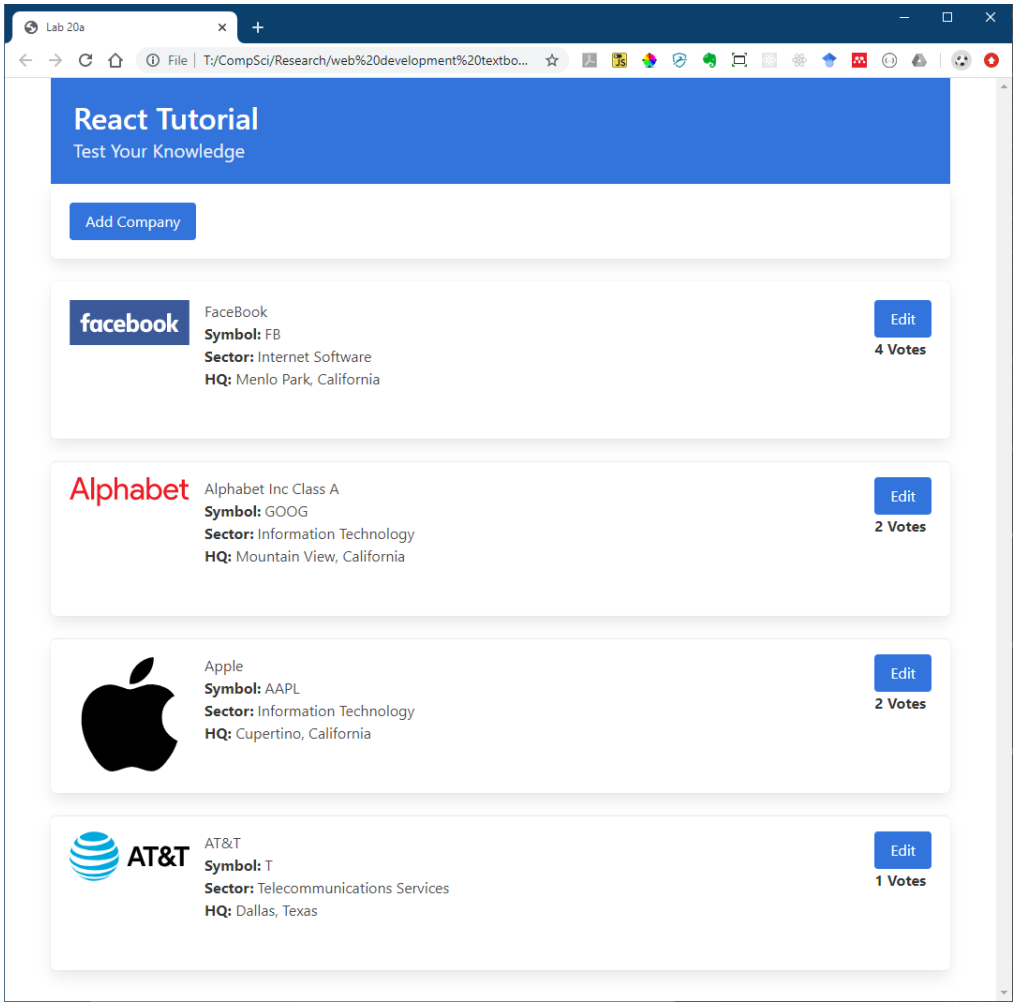


*Figure 20a.8 – Completed Test Your Knowledge #3 in normal mode*

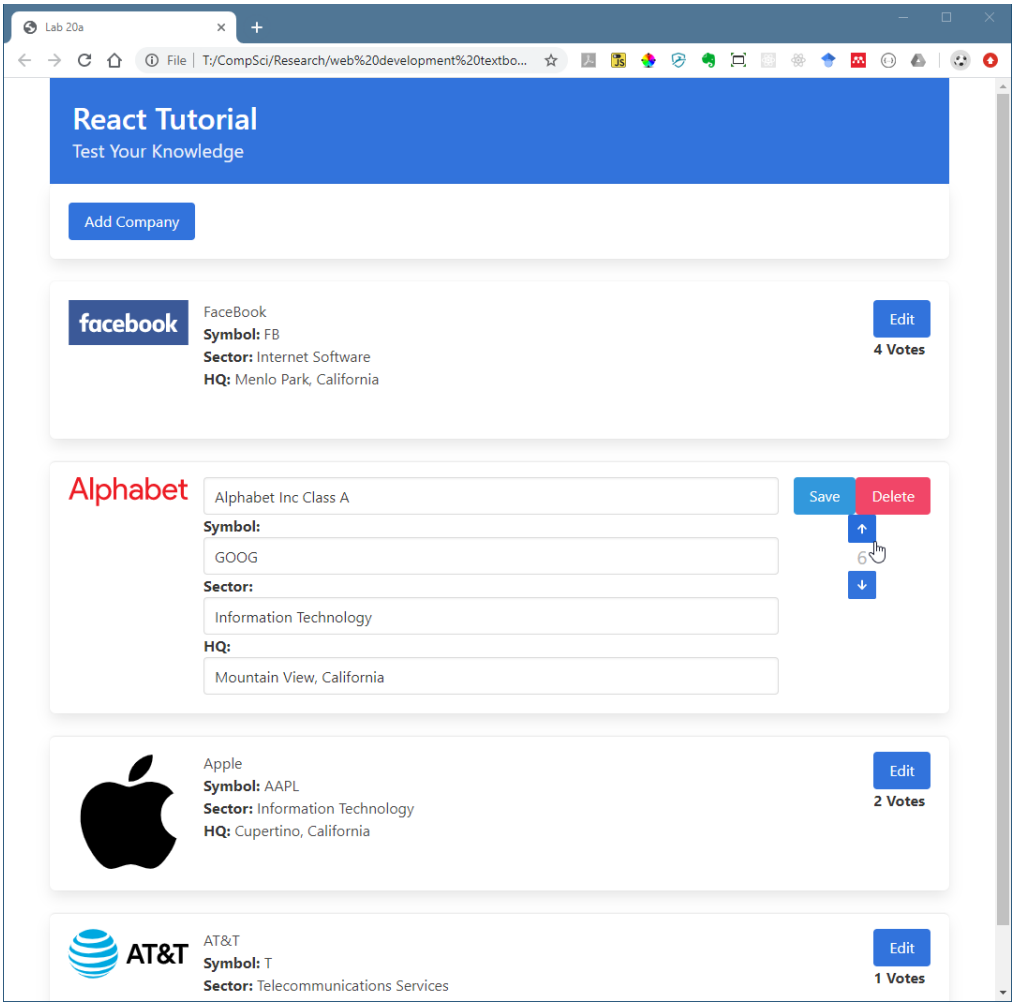


*Figure 20a.9 – Completed Test Your Knowledge #3 in edit mode*