LAB 11a

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

Note

This chapter's content has been split into three labs: Lab11a, Lab11b, Lab11c.

Approximate Time

The exercises in this lab should take approximately 120 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

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

Date Last Revised: January 16, 2023

Preparing Directories

1 The starting `lab11a` folder has been provided for you (within the zip folder downloaded from Gumroad).

*Note: these labs use the convention of `blue background` text to indicate filenames or folder names and **`bold red`** for content to be typed in by the student.*

Beginning React

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 11a.1 — Using JSX

1 Examine `lab11a-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 `lab11a-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 at the end of the document. 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
*/
const container = document.querySelector('#react-container');
ReactDOM.createRoot(container).render(<Company />);

</script>
```

Note that JSX is class sensitive and follows XML rules. Also, the class attribute has been changed to className.

5 Test in browser. NOTE: 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 11a.1.

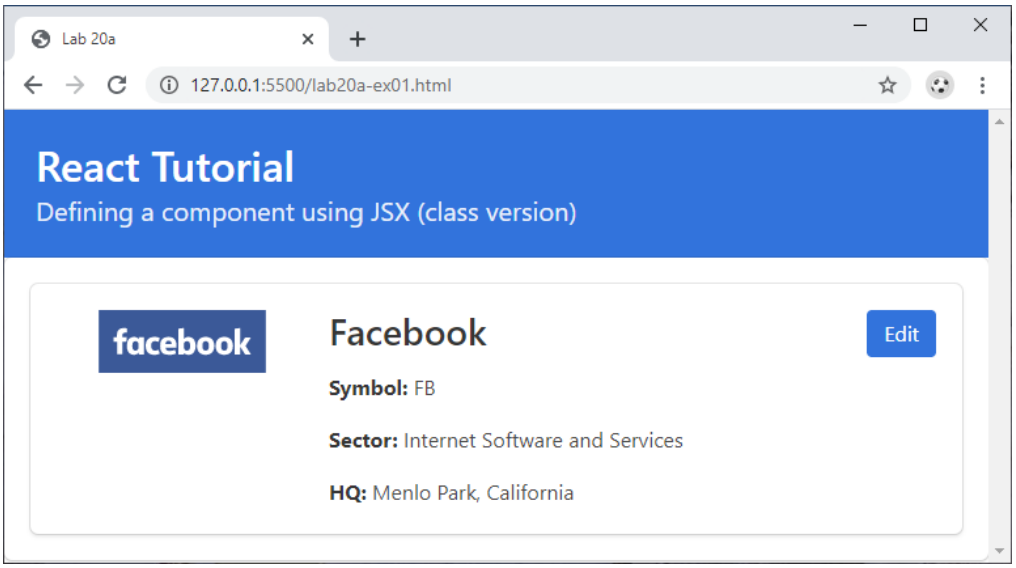

Figure 11a.1 – Finished Exercise 11a.1

Exercise 11a.2 — Functional Components

1 Open `lab11a-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 `createRoot.render` call as follows:

```
ReactDOM.createRoot(container).render(simple);
```

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 `createRoot.render` call as follows:

```
ReactDOM.createRoot(container).render(Simple());
```

Here we are passing the component returned from the Simple function.

7 Modify the `createRoot.render` call as follows:

```
ReactDOM.createRoot(container).render(<Simple />);
```

Note that functional components can be referenced as markup. This is a much more common way of working with React. Also, React expects functional components to begin with a capital letter.

8 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.

9 Add the following new component.

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

You will rarely do this: it is here just to illustrate what React is doing behind the scene.

10 Modify the `createRoot.render` call as follows:

```
ReactDOM.createRoot(container).render(alternate);
```

The Babel transcompiler (which converts from JSX to vanilla JavaScript) will convert the code from steps 7 and 8 into something similar to that shown in steps 9 and 10.

Exercise 11a.3 — Component Parameters

1 Open `lab11a-ex03.html` and modify the `ReactDOM.createRoot` call as follows (but don't test yet).

```
ReactDOM.createRoot(container)
   .render(<Simple title="Using Props" />);
```

2 Modify the component as follows and test.

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

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

4 Comment out the previous version of the component and add a new version (all on one line in your editor) and test.

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

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

Exercise 11a.4 — Class Components

1 Open `lab11a-ex04.html`.

Notice that it contains the class component from exercise 1.

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.

```
ReactDOM.createRoot(container).render(app);
```

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.createRoot(container).render(<App />);
```

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 most of the `<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>
    );
}
```

Notice that you will have to change the class attribute to classsName.

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 11a.2.

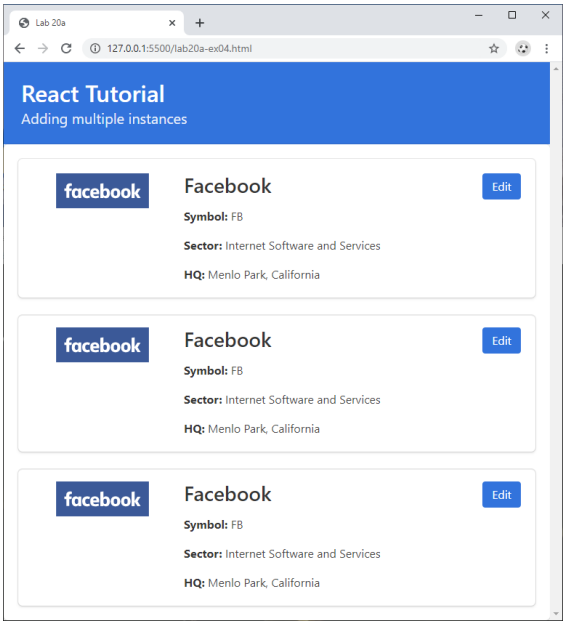

Figure 11a.2 – Finished Exercise 11a.4

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 11a.5 — Using Props

1 Open `lab11a-ex05.html`.

2 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 (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>
  );
};
```

3 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>
    );
  }
}
```

4 Modify the `App` component as shown in the following code:

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

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

5 Modify the `Company` 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>
   );
  }
}
```

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>;
```

Notice there is no return statement here: remember that with arrow syntax, no return is needed (i.e., it is implicit) if function consists of just a single executable line. Notice the required semicolon.

7 Modify the `App` component as shown in the following code:

```
const App = () => {
  return (
    <main className="container">
      ...
    </main>
  );
}
```

The point of this change is just to remind you that you can use both class components and functional components.

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 11a.6 — Adding Behaviors

1 Open `lab11a-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.

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 in via State. In the next example, you will add in your first State variable, which will be a flag indicating whether or not we are in edit mode.

Exercise 11a.7 — Adding State

1 Open `lab11a-ex07.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};
  }

  /* 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 11a.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 Comment out the constructor you created in step 1. Add the following and test:

```
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.

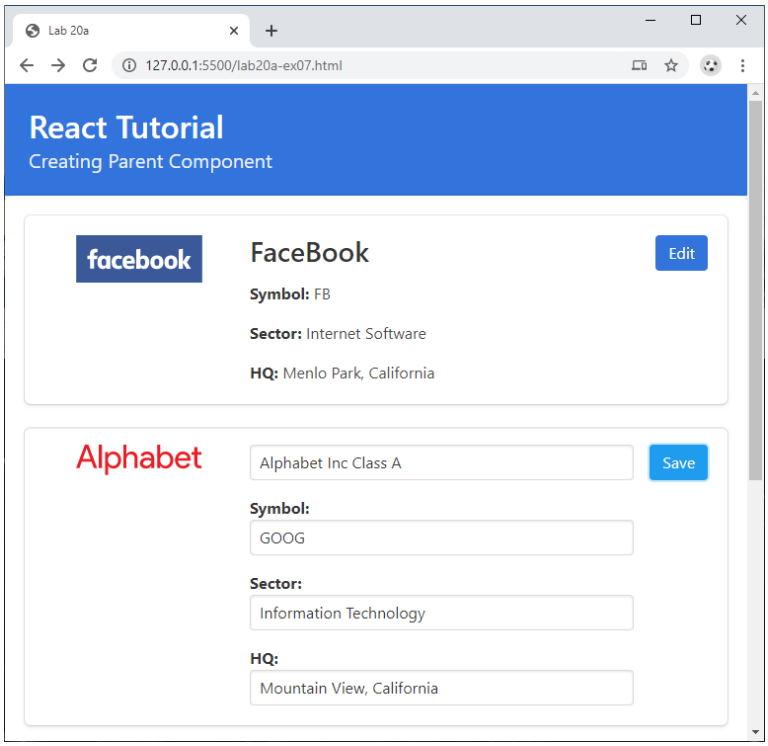

Figure 11b.3 – Finished Exercise 11a.07

In mid 2019, React introduced a new mechanism known as React Hooks. The most important of these hooks is a new way of working with state.

Exercise 11a.8 — Using Hooks for State

1 Open `lab11a-ex08.html`, add the following code, and test:

```
/* Notice that Company is a functional component now and not a class
component.

Hooks provides state capabilities to functional components */

const Company = (props) => {

   const editClick = () => {
      setEditing(true);
   };
   const saveClick = () => {
      setEditing(false);
   };

   const renderNormal = () => {
      return (
         ...
      );
   };


   const renderEdit = () => {
      return (
         ...
      );
   };

   /* add hook-based state code here */
   const [editing, setEditing] = React.useState(false);
   if (editing) {
      return renderEdit();
   }
   else {
      return renderNormal();
   }

}
```

Notice that Company is now a functional component and not a class component.

Exercise 20a.9 — Creating a Parent Component

1 Open `lab11a-ex09.html` and add the following code:

```
const Portfolio = (props) => {

  /* you can copy the data from the App function */
  const [companies, setCompanies] = React.useState(
    [
      {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"}
    ]
  );

  /* The render for this component will use map to loop through the
     data and generate the appropriate Company elements */
  return (
    <section className="content box">
      { companies.map( c => <Company symbol={c.symbol}
                                     sector={c.sector} hq={c.hq}>
                              {c.name}
                            </Company> ) }
    </section>
  );
}
```

This illustrates a very common coding idiom in React: namely, using the map() function on a data array in order to output multiple components. Later you will learn how to retrieve data from a Web API in React instead of using a hard-coded array.

2 Modify the App component as follows:

```
const App = () => {
  return (
        <main className="container">
          <Header subtitle="Creating Parent Components" />
          <Portfolio />
        </main>
      );
}
```

3 Test in browser.

The result in the browser should be the same as the previous exercise. However, take a look at the console. You will see a warning that each child in list needs a unique key.

4 Modify the `Portfolio` component as follows and test.

```
{ companies.map( (c, ind) => <Company symbol={c.symbol}
                                      sector={c.sector} hq={c.hq}
                                      key={ind} >
                                  {c.name}</Company> ) }
```

Here we are making use of the ability to add an index parameter to the `map()` function.

5 Add the following function to the `Portfolio` component. Don't test yet.

```
  const createCompany = (obj, index) => {
     return (<Company symbol={obj.symbol}
                      sector={obj.sector}
                      hq={obj.hq} key={index}
                      >{obj.name}</Company>)
  }
```

6 Modify the `Portfolio` component as follows and test.

```
{ companies.map( (c,ind) => createCompany(c,ind) ) }
```

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 11a.10 — Controlled Form Components

1 Examine `lab11a-ex10.html` and then initialize the state as follows:

```
const SampleForm = (props) => {
    // initialize our state
    const [company, setCompany] = React.useState(
        {
            name: "FunWebDev Corp",
            sector: "Textbooks",
            comments: "They know things!"
        }
    );
```

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

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

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

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

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

You will see that the form data isn't yet reflected in the state.

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

```
const handleCompanyChange = (e) => {
    const updatedCompany = { ... company };
    updatedCompany.name = e.target.value;
    setCompany(updatedCompany);
};
const handleSectorChange = (e) => {
    const updatedCompany = { ... company };
    updatedCompany.sector = e.target.value;
    setCompany(updatedCompany);
};
const handleCommentsChange = (e) => {
    const updatedCompany = { ... company };
    updatedCompany.comments = e.target.value;
    setCompany(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={company.name}
        onChange={handleCompanyChange}  />

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

<textarea className="textarea"
        value={company.comments}
        onChange={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 when there are a lot of form elements. 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={ handleChange }  />

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

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

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

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

This should work.

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`, as shown in the next exercise.

Exercise 11a.11 — Uncontrolled Form Components

1 Examine `lab11a-ex11.html` and then make the following modification:

```
const SampleForm = (props) => {
    // define some references to our DOM form elements
    const name = React.createRef();
    const sector = React.createRef();
    const comments = React.createRef();
}
```

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

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

<select ref={sector}>

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

3 Modify the handleSubmit method as follows:

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

COMPONENTS WORKING TOGETHER

The first exercise in this section will construct a user interface as a series of nested components. It will also display components based on data within an array.

Exercise 11a.12 —COMPONENT COMPOSITION

1 Examine `lab11a-ex12-markup.html` in the code editor and the browser. This provides the markup your application will be creating in React.

2 Open and examine `lab11a-ex12.html` in the code editor. Notice that it already contains a data array and a few components already defined. Begin by creating the following component.

```
const ListItem = props => {
   return (
      <li><a>{props.name}
        <button className="is-small is-light button">
           Move
        </button>
      </a></li>
   );
};
```

This component will render each list item in the source and destination columns.

3 Now add the following component which uses the previous one.

```
const Column = (props) => {
   return (
      <section className="column">
         <h2 className="is-size-5 has-text-centered">
           {props.heading}
         </h2>
         <ul className="menu-list">
            { props.data.map( m =>
               <ListItem name={m.title} key={m.id} id={m.id} /> )
            }
         </ul>
      </section>
   )
};
```

This component displays a column of movies. Notice that it uses the map() function to output a list item component for each element in the passed-in data array.

4 Add the following to the `App` component and test.

```
return (
   <main >
      <Header subtitle="Component Interaction" />
      <article className="columns">
         <Column heading="Source" data={movies} />
         <Column heading="Destination" data={movies} />
      </article>
   </main>
);
```

Because we are passing in the entire movies array to each column component, each column displays all the movies.

So far in this lab, you have learned about two types of component data in React: props and state. The key feature of state data is that while it can be altered by its component, it belongs to the component and is unavailable outside of the component. But what if two components want access to the same data? What if one component wants to display some data and another component wants to edit that same data? The most common approach is to let the parent handle the state needs of its child components. The next exercise illustrates this approach.

Exercise 11a.13 — Component Data Flow

1 You will continue modifying `lab11a-ex12.html` in this exercise.

For the finished version, we will want the button in the list item to move the clicked movie from one column to the other. This will require state data that is managed by the parent of the two `Column` components, which in this case is the `App` component.

2 Add the following to the App component, after the `movies` array:

```
const [source, setSource] = React.useState(movies);
const [destination, setDestination] = React.useState([]);
```

This uses the hooks approach to define two state variables: one to keep track of the movies in the source column, and one for movies in the destination column. In this case, all the movies are added to the source array, while the destination array is simply an empty array.

3 Modify the App component as follows and test.

```
return (
   <main >
      <Header subtitle="Component Interaction" />
      <article className="columns">
         <Column heading="Source" data={source} />
         <Column heading="Destination" data={destination} />
      </article>
   </main>
);
```

The second column should now display no movies. The next step will be to write functions that will modify these two state arrays. These functions will need to be defined in the same component as the state itself. In this case, this is the App component.

4 Add the following function to the App component.

```
// responsible for moving a movie from source to destination
const moveFromSourceToDestination = (id) => {
   // first find movie to move
   const movieTo = source.find (m => m.id == id);
   // create new array which doesn't contain that movie
   const newSource = source.filter( m => m.id != id );
   // update source state
   setSource(newSource);
   // add movie to destination
   destination.push(movieTo)
   // update destination source
   setDestination(destination);
};
```

5 Add the following function to the App component.

```
// responsible for moving a movie from destination to source
const moveFromDestinationToSource = (id) => {
   const movieTo = destination.find (m => m.id == id);
   const newDest = destination.filter( m => m.id != id );
   setDestination(newDest);
   source.push( movieTo)
   setSource(source);
};
```

Now that these functions are defined, they need to be passed down to the Column components that will use these functions in response to the button clicks.

6 Modify the `ListItem` component as follows and test by clicking the buttons.

```
return (
   <main >
      <Header subtitle="Component Interaction" />
      <article className="columns">
         <Column heading="Source" data={source}
             update={moveFromSourceToDestination} />
         <Column heading="Destination" data={destination}
             update={moveFromDestinationToSource} />
      </article>
   </main>
);
```

7 Modify the `Column` component as follows.

```
const Column = (props) => {
   return (
      <section className="column">
         <h2 className="is-size-5 has-text-centered">
           {props.heading}
         </h2>
         <ul className="menu-list">
            { props.data.map( m =>
               <ListItem name={m.title} key={m.id} id={m.id}
                         update={props.update} /> )
            }
         </ul>
      </section>
   )
};
```

The buttons that do the moving are in the ListItem component. Therefore, the update function needs to be passed down to the ListItem component.

8 Modify the `ListItem` component as follows then test by clicking the buttons.

```
const ListItem = props => {
   const handleClick = (e) => {
      alert("Movie clicked has id=" + props.id);
   };
   return (
      <li><a>{props.name}
        <button className="is-small is-light button"
                onClick={handleClick} >
           Move
        </button>
      </a></li>
   );
};
```

9 Modify the `ListItem` component as follows then test.

```
const ListItem = props => {
   const handleClick = (e) => {
      props.update(props.id);
   };
   return (
      <li><a>{props.name}
        <button className="is-small is-light button"
                onClick={handleClick} >
           Move
        </button>
      </a></li>
   );
};
```

You should now be able to move items between these two columns.

Now, let's go back to our earlier form example, where we will preserve user data changes via React state.

Exercise 11a.14 — Preserving Form Changes

1 Open `lab11a-ex14.html` and modify the constructor method of the `Company` class (some code omitted):

```
const Company = (props) => {
   const inputName = React.createRef();
   const inputSymbol = React.createRef();
   const inputSector = React.createRef();
   const 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={saveClick}>Save</button>
          <button className="button is-danger"
                  onClick={deleteClick} >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 (
      ...
      <div className="media-content">
          <h2><input type="text" className="input"
                     defaultValue={props.children}
                     ref={inputName} /></h2>
          <p><strong>Symbol:</strong>
             <input type="text" className="input"
                    defaultValue={props.symbol}
                    ref={inputSymbol} /></p>
          <p><strong>Sector:</strong>
             <input type="text" className="input"
                    defaultValue={props.sector}
                    ref={inputSector} /></p>
          <p><strong>HQ:</strong>
             <input type="text" className="input"
                    defaultValue={props.hq}
                    ref={inputHQ} /></p>
      </div>
      ...
```

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

```
/* notice that the parent is responsible for making changes to the
   State of its children  */
const saveCompany = (newName,newSymbol,newSector,newHq,index) => {
   const tempArray = [...companies];
   tempArray[index] = { name: newName, symbol: newSymbol,
                        sector: newSector, hq: newHq };
   setCompanies(tempArray);
}

const deleteCompany = (index) => {
    const tempArray = [...companies];
    tempArray.splice(index,1);
    setCompanies( tempArray );
}
```

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

```
const createCompany = (obj, index) => {
    return (<Company symbol={obj.symbol}
                     sector={obj.sector}
                     hq={obj.hq} key={index} index={index}
                      saveData={saveCompany}
                      removeData={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 make the `saveClick` and `deleteClick` event handlers for the buttons 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  */
const saveClick = () => {
  /* retrieve the user input */
  let newName = inputName.current.value;
  let newSymbol = inputSymbol.current.value;
  let newSector = inputSector.current.value;
  let newHq = inputHQ.current.value;

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

  setEditing(false);
};
const deleteClick = () => {
      props.removeData(props.index);
      setEditing(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).

TEST YOUR KNOWLEDGE #1

1 The instructions for this Test Your Knowledge are on page 575-6 of Chapter 11. The finished result can be seen in Figure 11a.5. The form changes will appear in the list as they are made by the user.

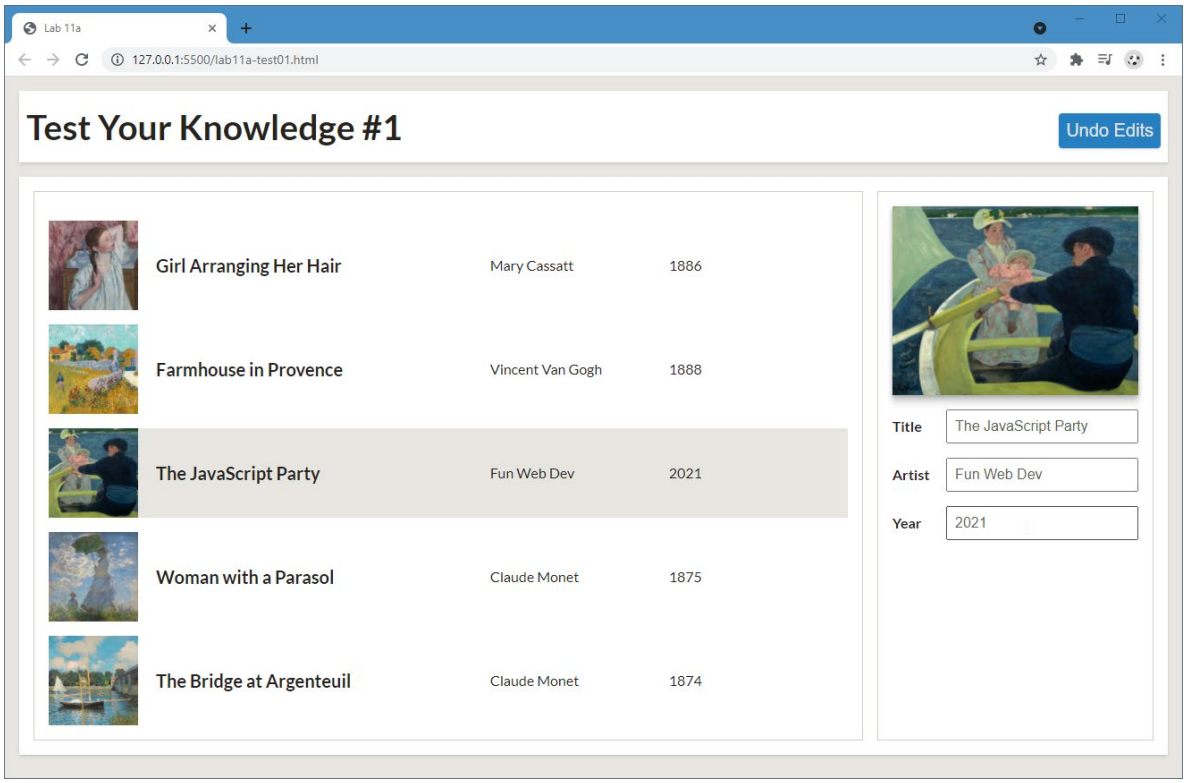

Figure 11a.5 – Finished Test Your Knowledge #1

TEST YOUR KNOWLEDGE #2

1. Open `lab11a-test02-markup-only.html` in the browser.

   This provides the markup needed for the exercise.

2. You will be starting with `lab11a-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 `lab11a-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 `lab11a-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 11a.7. 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 11a.8.

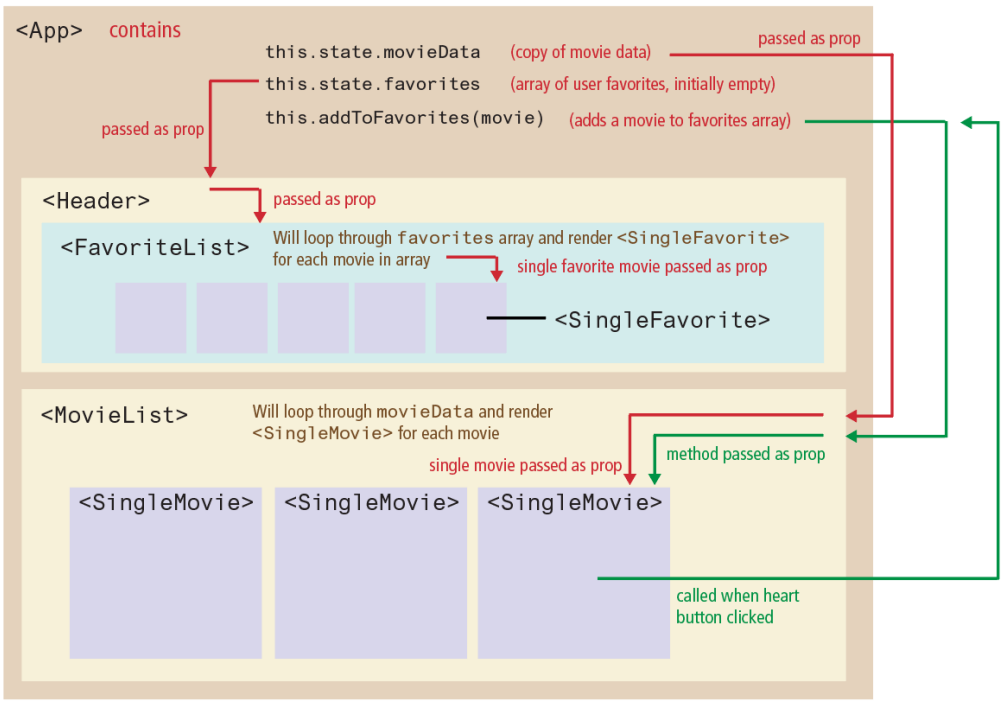

Figure 11a.7 – Data flow between components

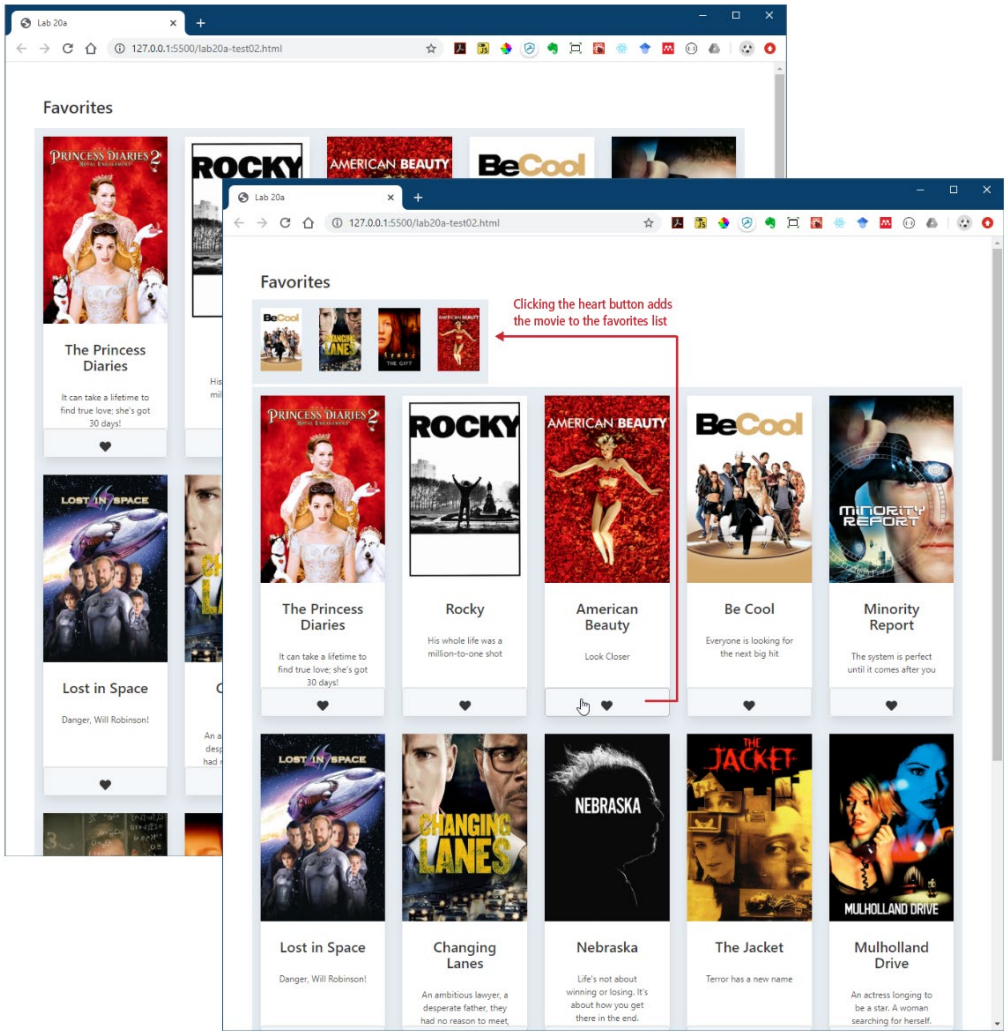

Figure 11a.8 – Finished Test Your Knowledge #2