LAB 11b

INTERMEDIATE REACT

What You Will Learn

- How to use a build tool (Vite) to setup a React application.
- How to interact with the React lifecycle methods
- How to implement data flow with functional and class components
- How to implement program flow with React Router

Note

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

Approximate Time

The exercises in this lab should take approximately 60 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

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

Date Last Revised: March 3, 2024
Revised: Modified Exercises 11b.4. 11b.5, and 11b.12. Added some headings.

Preparing Directories

1 The starting `lab11b` folder has been provided for you (within the zip folder downloaded from Gumroad). **You will add content from this folder at various points in this lab.**

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.

React Build Tools

In this lab, you will use the build tool approach to construct a React application. That is, instead of using a babel transpiler at run-time to convert from JSX to JavaScript, you will a command-line tool that you run at design-time while developing. In the past, the preferred tool for this was the popular `create-react-app` application; but in the past few years, developers are instead using the `Vite` application, which we will use in this lab. These build tools do make use of `npm` and `node`, both of which you will have to install or use an environment that already has it installed.

This lab will also cover key React concepts as the previous lab, but add routing and extracting data from an external API.

Exercise 11b.1 — Installing Node

1 The mechanisms for installing Node vary based on the operating system. You can find installers at `https://nodejs.org/`.

If you wish to run Node locally on a Windows-based development machine, you will need to download and run the Windows installer from the Node.js website.

If you want to run Node locally on a Mac, then you will have to download and run the install package.

If you want to run Node on a Linux-based environment, you will likely have to run curl and sudo commands to do so. The Node website provides instructions for most Linux environments.

If you are using a cloud-based development environment such as Github Codespaces, Node is likely already available in your workspace.

2 To run Node, you will need to use Terminal/Bash/Command Window. Verify it is working after you have finished the installation by typing the following commands:

```
node -v
npm -v
npx -v
```

The second command will display the version number of npm, the Node Package Manager which is part of the Node install. The third command (npx) is newer and might not be on your system: it is a tool for executing Node packages (though you can do so also via npm).

3 Navigate to the folder you are going to use for your source files in this lab.

4 Create a simple file from the command line via the following command:

```
echo "console.log('hello world')" > hello.js
```

5 Verify your node install works by running this file in node via the following command:

```
node hello.js
```

Not the most amazing program but you will do more exciting things later with Node (in lab 13).

Exercise 11b.2 — Installing Vite

1 Using your terminal, ensure you are in your `lab11b` folder.

2 Run the following command:

```
npm create vite@latest lab11b-react-app -- --template react
```

The extra double-dash is needed. This will create a scaffolded react application in a folder named lab11b-react-app. It will also install the create-vite application the first time it is run.

3 Switch to the `lab11b-react-app` folder via:

```
cd lab11b-react-app
```

4 Examine the folders created by `vite`. Let's spend a few moments examining them.

Examine the `index.html` file.

The projects created with `vite` (and `create-react-app` in the past) are SPA (single-page applications). This file is that one page, and contains the `root` element that all React components will be rendered into.

5 Examine the `main.jsx` file in the `src` folder.

The code you will write for this project will be within this `src` folder. The `main.js` file renders the `<App>` element to the root element that we just looked at in `index.html`.

6 Examine the `App.js` file in the `src` folder. We will be replacing this sample content eventually, but for now, we will simply look at it. Notice that it uses `import` statements to create references to other functions or classes that are part of the React infrastructure. In this lab, you will be creating a number of new components; each one will exist, like this one, in its own file.

7 Examine the `package.json` file in the `lab11b-react-app` folder.

Notice that it provides both development and production build commands, two source dependencies, and a variety of developer dependencies. Developer dependencies are necessary (or maybe just helpful) for the developer, but won't be part of the build version hosted on the server. The regular dependencies are however necessary for the application to run. These dependencies will be added in the next step.

8 Run the following command:

```
npm install
```

This will take a few minutes as it will download and install all the dependencies listed in the `package.json` file.

9 Examine the `node_modules` folder.

There will be dozens of folders, each containing JavaScript files. This folder is used to contain source code downloaded from the npm site. Some of this code is used to run the create-react-app infrastructure; some of it is used by React. The builder will combine and include any JavaScript in this folder needed for production.

10 To run and test our project, you need to switch to the `my-app` folder and run the project. This will start its own development web server (using node). To do this, run the following command from the terminal:

```
npm run dev
```

This should display a message saying compilation was successful (see Figure 11b-1). It should also automatically open a browser tab with the sample starting react app visible. You may also see warning messages about features that will be deprecated in the future. You can ignore these.

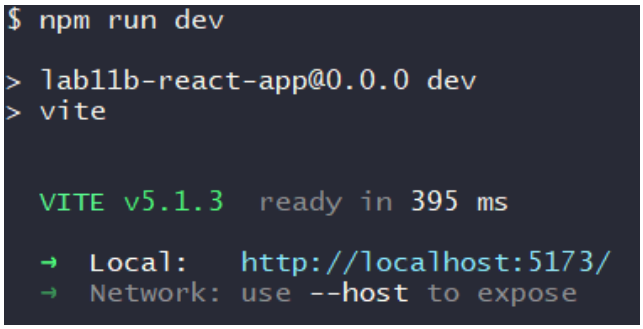

Figure 11b.1 – Running the application

11 Copy the displayed local URL in the terminal and paste it into your browser. Click on the count button/label to increment the value.

When everything works correctly, you should see the something similar to that shown in Figure 11b.2.

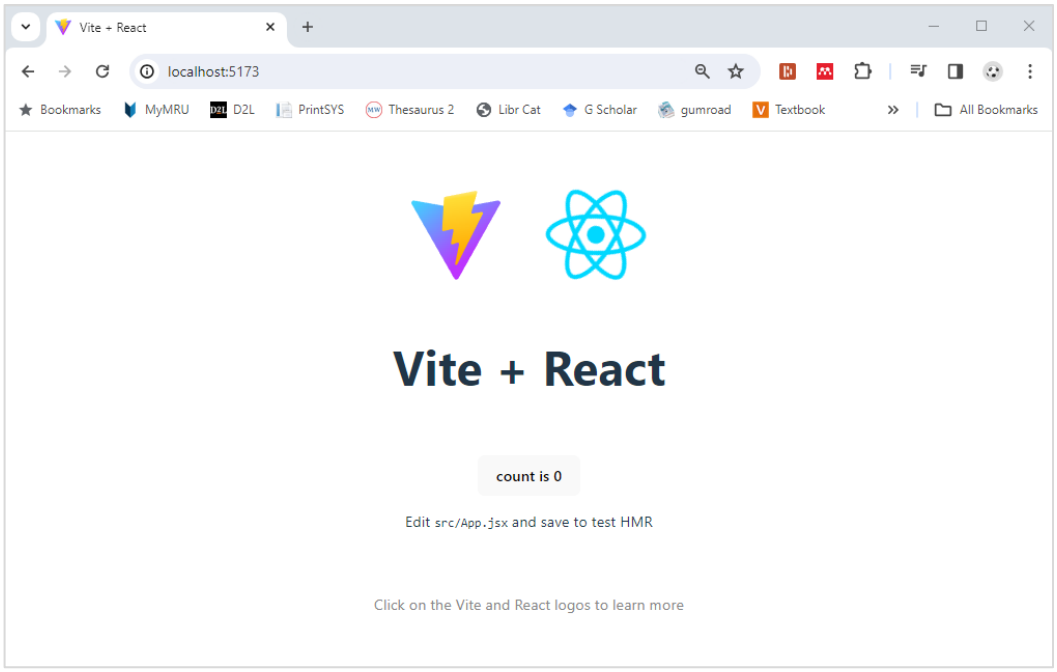

Figure 11b.2 – Finished Exercise 11b.02

CREATING COMPONENTS

Exercise 11b.3 — Creating Functional Components

1. In the `src` folder, create a folder named `components`.

   In this folder, you will be placing the components you create.

2. Create a new file in this new folder named `HeaderBar.jsx`.

3. Add the following code to this file and save:

```
const HeaderBar = function (props) {
  return (
        <div className="header-titles">
            <h1>Travel Image App</h1>
            <p>Using Vite</p>
        </div>
  );
}

export default HeaderBar;
```

   Notice the last line. This is needed for making this component available to our other components. What you are creating here then is a JavaScript module; you may want to review pages 493-7 in Chapter 10.

4 Create a new file in this same folder named `HeaderMenu.jsx` as follows:

```
const HeaderMenu = function (props) {
  return (
      <nav>
          <button>About</button>
          <button>Upload</button>
          <button>Download</button>
      </nav>
  );
}
export default HeaderMenu;
```

5 Create a new file in this same folder named `HeaderApp.jsx` as follows:

```
import HeaderBar from './HeaderBar.js';
import HeaderMenu from './HeaderMenu.js';

const HeaderApp = (props) => {
    return (
        <header className="header">
          <HeaderBar />
          <HeaderMenu />
        </header>
        );
}
export default HeaderApp;
```

To use another component, you have to provide the appropriate import statement. Notice here that you have added import references to the two components you just created.

6 Modify the `App.js` file as follows (notice some lines in the original are to be deleted):

```
import logo from './logo.svg';
import './App.css';
import HeaderApp from './components/HeaderApp.jsx';

function App() {
    return (
      <main>
        <HeaderApp />
      </main>
    );
  }
}
export default App;
```

7 Save this file. It should compile correctly and you should be able to view it in your browser.

8 If you wanted to make use of an external CSS library such as Bootstrap, you can add it to the `<head>` of the file `index.html`. Here we are going to add the Bulma CSS library:

```
<meta name="viewport" content="width=device-width …">
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bulma@0.9.2/css/bulma.min.css">
```

9 Locate the `index-styles-to-copy.css` file in the starting files that have been provided to you as part of this lab. This file contains a few dozen already-created styles. Copy the content in this file and replace the content of the `index.css` file in the `src` folder with it. Test in browser.

The result should look similar to that shown in Figure 11b.3.

10 Move the files in the `images` folder in the starting files into the `public` folder created by Vite.

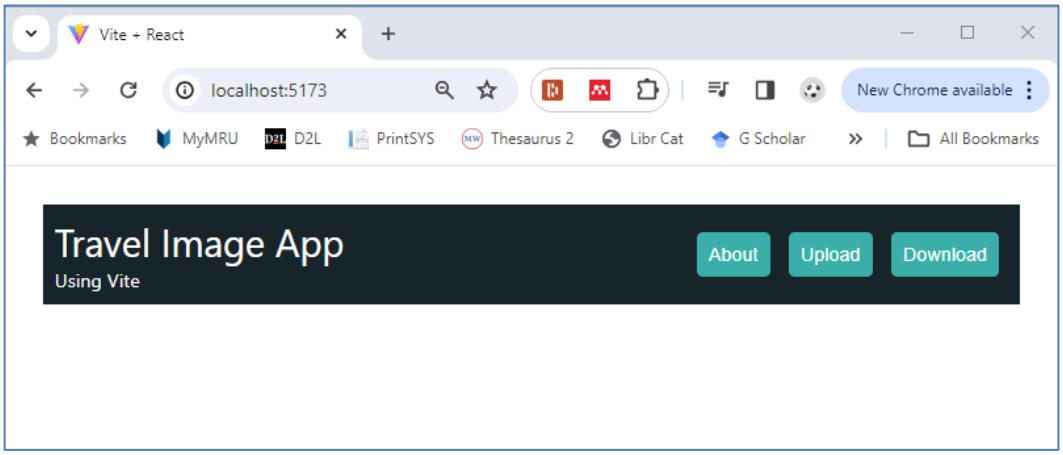

Figure 11b.3 – Finished Exercise 11b.3

Exercise 11b.4 — Adding Behavior to a Component

1 Locate the `Company.jsx` file in the starting files that have been provided to you as part of this lab and move it into the `components` folder.

If you examine this file, you will see that it is similar to the Company component from Lab11a but set up using the module approach.

2 Locate the `CompanyLogo.jsx` file in the starting files that have been provided to you as part of this lab and move it into the `components` folder.

3 Locate the `App.jsx` file in the starting files that have been provided to you as part of this lab and move it (replace the exist `App.jsx` file) into the `src` folder.

If you examine this file, you will see that it is similar to the App component from Lab11a.

4 Test in browser. It should look similar to that shown in Figure 11b.4

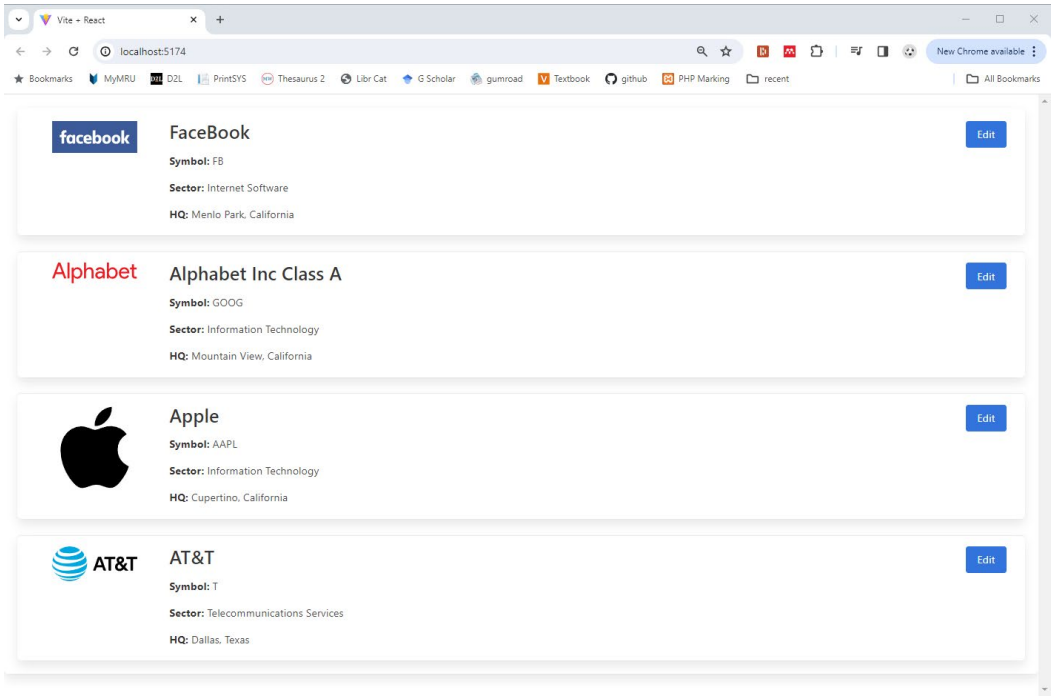

Figure 11b.4 – Finished Exercise 11b.4

5 Test in browser. It should look similar to that shown in Figure 11b.4.

6 Add the following code to your `Company` component (some code omitted) and test.

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

   const edit = () => {
      alert("now editing " + props.data.name);
   };

   return (
     ...
   <button className="button is-link" onClick={edit}>Edit</button>
```

7 Modify the code as follows and test.

```
<button className="button is-link"
        onClick={ () => { alert("this is allowed also")} }>
```

While helpful, props are read-only, meaning that a component can't change them. Components often need data that it can manipulate and change. This is achieved in via State. Each component can have its own 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 11b.5 — Working with Component State

1 Add the following code to `Company`.

```
import { useState } from 'react';

const Company = (props) => {
   const [editing, setEditing] = useState(false);

   const edit = () => {
      setEditing(true);
   };
   const save = () => {
      setEditing(false);
   };
```

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. Prior to Hooks, the only way to access component state was via class components. The useState function is passed the initial value of our state value. It can be any object, but here we are using a Boolean to indicate whether the component is in edit mode. The function returns an array with two elements: a variable containing the current value of the state variable and a function for changing it. The edit and save functions will change the state variable.

2 Add the following code:

```
const renderEdit = () => {
   return (
      <article className="box media ">
         <CompanyLogo symbol={props.data.symbol} />
         <div className="media-content">
         <h2><input type="text" className="input"
                  defaultValue={props.data.name} /></h2>
         <p><strong>Symbol:</strong> {props.data.symbol}</p>
         <p><strong>Sector:</strong>
            <input type="text" className="input"
               defaultValue={props.data.sector} /></p>
         <p><strong>HQ:</strong>
            <input type="text" className="input"
                     defaultValue={props.data.hq} /></p>
         </div>
         <div className="media-right">
            <button className="button is-info" onClick={save}>
            Save</button>
         </div>
      </article>
   );
};
```

When the company is in edit mode, it will display form elements.

3 Modify the code as follows:

```
if (editing) {
   return renderEdit();
}
else {
   return renderNormal();
}
```

The Edit button should change the rendering of the component. 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, which makes sense given the data is currently passed to the component in the read-only props. To do so will require learning how to pass data between components. But before that, we need to learn more about forms in React.

4 Modify the code as follows:

```
const save = () => {
   setEditing(false);
   let temp = count + 1;
   setCount(temp);
};

const [count, setCount] = useState(0);
console.log("render for " + props.data.name + " count="+count);
...
<h2>{props.data.name} {count}</h2>
```

5 Test with the console visible (or display console and refresh). Repeatedly click the edit then save buttons.

Notice how a component function gets called twice when rendered. Since there are four `Company` components, you should initially see 8 render messages in the console.

Notice that when you click the Save button, only that one component is re-rendered.

6 Modify the code as follows and test.

```
const save = () => {
   setEditing(false);
   let temp = count + 1;
   setCount(temp);
   console.log('temp='+temp+' count='+count);
};
```

This should surprise/baffle you, since you will likely see a different value for `count` in the `save()` function, but the correct value when the component is rendered. When you make a call to a state setter, the actual state change is only scheduled rather than immediately changed.

7 Add the following to the top of `App.jsx`:

```
import { useState } from 'react';
```

8 Modify `App.jsx` as follows, then test by clicking the new change button. Be sure to examine the console.

```
const [change, setChange] = useState(0);
const handler = () => {
  let temp = change + 1;
  setChange(temp);
  comps[0].name = temp;
  console.log(comps[0].name);
}
console.log("**** App render change="+change+" "+comps[0].name);

return (
  <section  className="content box">
    <button className="button is-success" onClick={handler}>
        Change = {change}</button>
```

Notice that while the Change button should change the name of the first company from "Facebook" to a number (e.g., "1"), the Company component doesn't display the changed array value, even though it has re-rendered (according to your console messages). Figure 11b.5 illustrates this process.

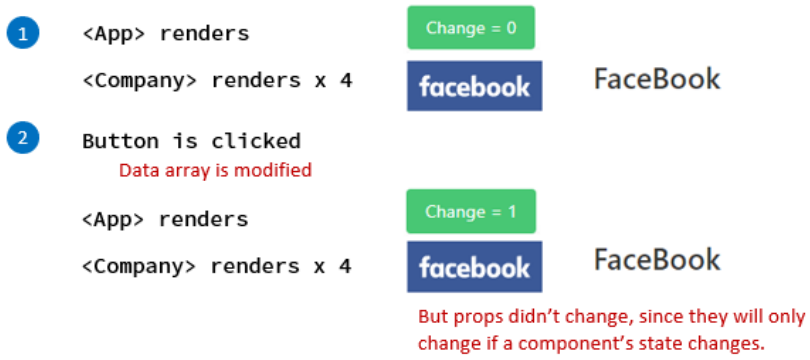

Figure 11b.5 – Relationship between state, props, and rendering

9 Modify `App.jsx` by putting the `comps` data array into state as follows (some code omitted):

```
const [comps, setComps]  = useState(
  [
      {name: "FaceBook", symbol: "FB",
        sector: "Internet Software",
        hq: "Menlo Park, California",
        description: "Description for Facebook"},
      ...
  ]
);
```

10 Modify the `handler` function as follows:

```
const handler = () => {
  let temp = change + 1;
  setChange(temp);
  // make a copy of the comps array
  const comp = [...comps];
  // modify the copy
  copy[0].name = temp;
  // change the state to use this altered copy
  setComps(tempArray);
}
```

11 Test. It now works as expected.

Moral of the story: if you want to update a variable, put it into a component's state.

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 11b.6 — CONTROLLED FORM COMPONENTS

1 Locate the `ControlledForm.jsx` file in the starting files that have been provided to you as part of this lab and move it into the `components` folder.

2 Modify `App.js` as follows (notice the commenting out of the first line):

```
//import Company from './components/Company.jsx';
import ControlledForm from './components/ControlledForm.jsx';
```

3 Comment out the current return statement in `App.js` and replace it with the following:

```
return ( <ControlledForm />);
```

4 Test.

It should display a form. You will be adding functionality to it.

5 Modify `ControlledForm.jsx` as follows:

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

  /* add additional functions here */
    const handleSubmit = (e) => {
        e.preventDefault();
        let values = `Current values are
              ${company.name}
              ${company.sector}
              ${company.comments}`;
        alert(values);
    };
```

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

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

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

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

8 Add the following methods to the `ControlledForm` 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.

9 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>
```

10 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 step provides an alternative.

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

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

12 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, and now requires only a single handler function.

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 11b.7 — Uncontrolled Form Components

1. Locate the `UncontrolledForm.jsx` file in the starting files that have been provided to you as part of this lab and move it into the `components` folder.

2. Modify `App.js` as follows (notice the commenting out of the first line):

```
//import Company from './components/Company.jsx';
//import ControlledForm from './components/ControlledForm.jsx';
import UncontrolledForm from './components/UncontrolledForm.jsx';
```

3. Modify the return statement in `App.js` with the following and test.

```
return ( <UncontrolledForm />);
```

4. Modify `UncontrolledForm.jsx` as follows:

```
import { useState, createRef } from 'react';

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

5. 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>
```

6. 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);
}
```

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

   The alert data should now reflect the current form data. Notice that in the uncontrolled version, you did not need to control what happens when each form element changed (that is, you did not need to write a OnChange handler for each input element, like you did in the Controlled Form example (Exercise 11b.6).

Components Working Together

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 demonstrates this approach; Figure 11b.6 illustrates its event and data flow.

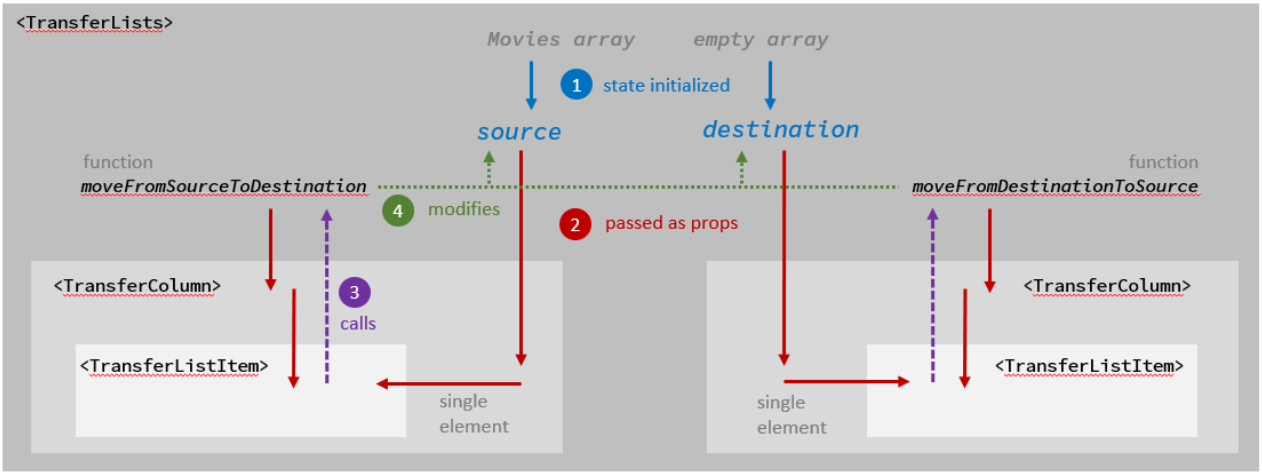

Figure 11b.6 – Event and data flow

Exercise 11b.8 — Component Data Flow

1. Examine `lab11b-ex08-markup-only.html` in the code editor and the browser. This provides the markup your application will be creating in React.
2. Locate the `TransferLists.jsx` and `TutorialHeader.jsx` files in the starting files that have been provided to you as part of this lab and move it into the `components` folder.
3. Create the following component in the `components` folder.

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

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

4 Create the following component in the `components` folder.

```
import TransferListItem from './TransferListItem.jsx';

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

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

5 Modify `TransferLists.jsx` as follows (some code omitted) and test.

```
import TutorialHeader from './TutorialHeader.jsx';
import TransferColumn from './TransferColumn.jsx';

const TransferLists = (props) => {
   ...
         <article className="columns">
            <TransferColumn heading="Source" data={movies} />
            <TransferColumn heading="Destination" data={movies} />
         </article>
```

Because we are passing in the entire movies data array to each column component, each column displays all the movies. 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 TransferColumn components, which in this case is the TransferLists component.

6 Add the following to the `TransferLists` component, after the `movies` array:

```
const [source, setSource] = useState(movies);
const [destination, setDestination] = 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 initially simply an empty array.

7 Modify `TransferLists.jsx` as follows and test.

```
<TransferColumn heading="Source" data={source} />
<TransferColumn heading="Destination" data={destination} />
```

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 TransferLists component.

8 Add the following functions to the `TransferLists` component (after the data array).

```
// 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);

};

// 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 `TransferColumn` *components that will use these functions in response to the button clicks.*

9 Modify `TransferLists.jsx` as follows and test.

```
<TransferColumn heading="Source" data={source}
    update={moveFromSourceToDestination} />
<TransferColumn heading="Destination" data={destination}
    update={moveFromDestinationToSource} />
```

Moving still doesn't work because the TransferListItem component needs these methods.

10 Modify the `TransferColumn` component as follows.

```
<ul className="menu-list">
   { props.data.map( m =>
      <TransferListItem name={m.title} key={m.id} id={m.id}
          update={props.update}  /> )
   }
</ul>
```

11 Modify the `TransferListItem` component as follows and then test.

```
const TransferListItem = (props) => {
   const handleClick = (e) => {
      props.update(props.id);
   };

   return (
      <li><a>{props.name}
        <button className="is-small is-light button"
            onClick={handleClick}>
```

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

CREATING A TAILWIND-BASED PROJECT

Tailwind CSS has become an essential part of many React developers' workflow. The next several exercises enhance a Tailwind-based React project.

Exercise 11b.9 — SETTING UP TAILWIND IN REACT

1. Using your terminal, ensure you are in your `lab11b-react-app` folder.
2. Enter the following commands:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

This install tailwindcss and its peer dependencies and generates your tailwind.config.js and postcss.config.js files.

3. Edit the just-generated `tailwind.config.js` as follows:

```
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

4. Comment out the existing styles in the `index.css` file inside the `src` folder and then add the following to the top of the file (before the comments).

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

This will of course mess up the styling of your previous exercises.

5. In `App.jsx`, comment out the previous return code and replace it with the following.

```
/*   return( <TransferLists /> ); */
return (
  <h1 className="text-3xl font-bold text-yellow-600 m-4">
    Hello world!
  </h1>
);
```

6. In `index.html`, comment out (or remove) the `<link>` element that references the bulma css library.
7. Run the Vite build process via:

```
npm run dev
```

You should see the hello world text with a bold yellow color. If so, then Tailwind is online!

Exercise 11b.10 — Setting Up the Component Structure

1 Examine `lab11b-ex10-markup-only.html` in the code editor and the browser. This previews the markup your application will be creating in React.

You will be adding code that will allow the user to select a company (using the pencil icon button), and then edit its fields.

2 Locate the `CompanyList.jsx`, `CompanyListItem.jsx`, `CompanyLogo.jsx`, `CompanyBrowser.jsx`, `CompanyForm.jsx`, and `SectorSelector.jsx` files in the starting files that have been provided to you as part of this lab and move them into the `components` folder.

These components are using tailwind css. This does make the markup quite busy for the purposes of the remaining exercises, so it will omit showing the classList attribute. In this exercise, you will be making changes from the outermost components to the innermost; the component is shown in Figure 11b.7.

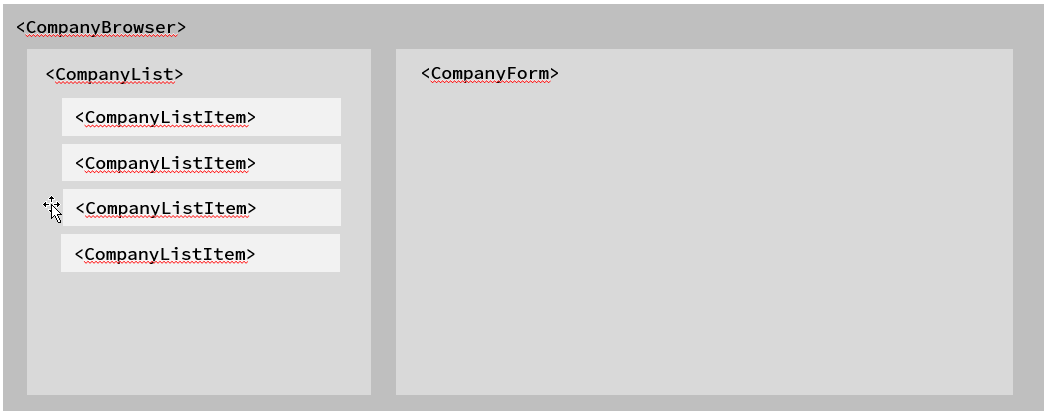

Figure 11b.7 – Component Structure

3 Make the following changes to the `App.jsx` file:

```
import CompanyBrowser from './components/CompanyBrowser.jsx';

function App() {
//   return( <TransferLists /> );
   return ( <CompanyBrowser /> );
}
```

4 Make the following changes to the `CompanyBrowser.jsx` file (some code omitted):

```
return (
   ...
   <CompanyList data={companies} />
   ...
```

Notice that the `companies` array is being stored in React state. In order to save changes made by the user in the form, the data must be in a component's state. Here you are

passing the companies array to the `CompanyList` *component, which will display each company in a* `CompanyListItem`*.*

5 Make the following changes to the `CompanyList.jsx` file (some code omitted):

```
<ul className="text-gray-300 ">
   { props.data.map( (c,indx) => <CompanyListItem
     company={c} key={indx} />) }
</ul>
```

6 Modify the markup returned in `CompanyListItem.jsx` as follows:

```
<div className="flex items-center">
   <span className="ml-3 font-medium">{props.company.name}</span>
</div>
<div>
   <button className="..." onClick={handleClick}  >
      <img src={editIcon}  className="w-5" />
   </button>
</div>
```

7 Add this function to `CompanyListItem.jsx`.

```
const handleClick = () => {
   alert("You clicked " + props.company.symbol);
}
```

8 Test in browser.

The list of companies should now appear along with the form. Test the pencil icon buttons.

Recall in Exercise 11b.8, changes to data that is shared between components has to happen in the parent component. As shown in Figure 11b.7, the parent component is `CompanyBrowser`, which indeed holds the array of companies in its state, so it will contain the functions that modify its state. The next exercise will add those functions and pass them to its two child components, as shown in Figure 11b.8.

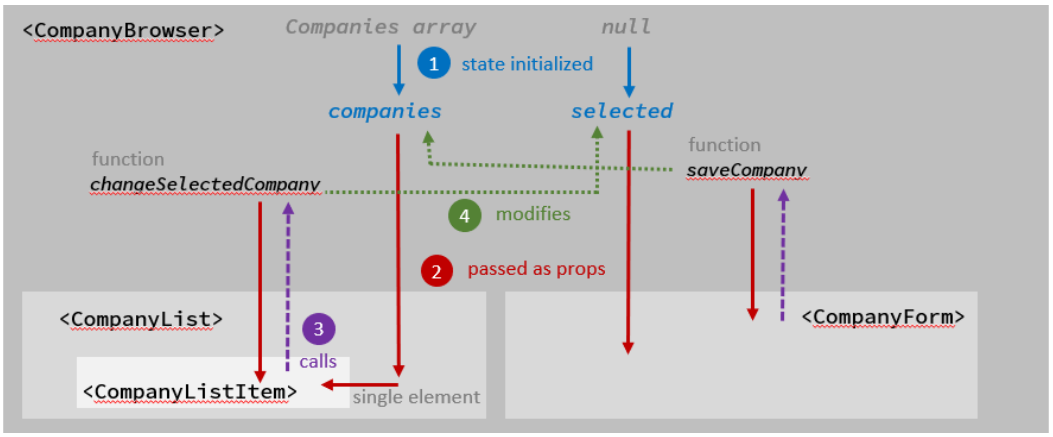

Figure 11b.8 – Event and data flow

Exercise 11b.11 — Adding in the State Management

1 Make the following changes to the `CompanyBrowser.jsx` file (some code omitted):

```
// indicates the selected company being edited in the form
const [selected, setSelected] = useState(null);
// changes the selected company
const changeSelectedCompany = (symbol) => {
   const sel = companies.find(c => c.symbol == symbol);
   setSelected(sel)
}
// updates state to reflect changes made in editing form
const saveCompany = (comp) => {
   const newCompanies = [...companies];
   const indx =
     newCompanies.findIndex(c => c.symbol == comp.symbol);
   if (indx >= 0) newCompanies[indx] = comp;
   setCompanies(newCompanies);
   setSelected(comp);
}
```

2 Make the following changes to returned markup in `CompanyBrowser.jsx`:

```
<div className="col-span-2 ">
   <SectorSelector />
   <CompanyList data={companies} changeCompany={changeSelectedCompany} />
</div>

<CompanyForm data={selected} save={saveCompany} />
```

3 Make the following changes to the `CompanyList.jsx` file:

```
<ul className="text-gray-300 ">
   { props.data.map( (c,indx) => <CompanyListItem
     company={c} key={indx} changeCompany={props.changeCompany} />) }
</ul>
```

Since the `CompanyListItem` component contains the edit icon button, you must pass the `changeCompany` function down to it.

4 Make the following change to the `CompanyListItem.jsx` file:

```
const handleClick = () => {
   props.changeCompany(props.company.symbol);
}
```

5 Make the following changes to the `CompanyForm.jsx` file:

```
import { useState } from 'react';

const CompanyForm = (props) => {

   // if no company has been selected, don't display the edit form
   if (! props.data) return;

   // this will be a controlled form: every time an element is changed
   // by the user, update the state data
   const handleChange = (e) => {
      // create a copy of the company object
      const updatedCompany = { ...props.data };
      // use bracket notation to modify the relevant object property
      updatedCompany[e.target.name] = e.target.value;
      // use the passed save function (implemented in CompanyBrowser)
      props.save(updatedCompany);
   }
```

3 Make the following changes to returned markup in `CompanyForm.jsx`:

```
<label htmlFor="name">Name</label>
<input type="text" name="name" value={props.data.name}
     onChange={handleChange} className= ... />

<label htmlFor="symbol">Symbol</label>
<input type="text" value={props.data.symbol} readOnly className= ... />

<label htmlFor="hq">HQ</label>
<input type="text" name="hq" value={props.data.hq}
     onChange={handleChange} className= .../>

<label htmlFor="sector">Sector</label>
<input type="text" name="sector" value={props.data.sector}
     onChange={handleChange} className= ...  />

<label htmlFor="description">Description</label>
<input type="text" name="description" value={props.data.description}
     onChange={handleChange} className= ...  />
```

4 Test. Click on the pencil icon for items in the list (the form data should change). Edit the form data. It will be persisted.

INTEGRATING EXTERNAL DATA

Most React applications need data. This data is usually fetched from an external API. In React, we have to retrieve data at a specific point in the component's life cycle. Prior to React Hooks (circa 2019), this was done during the `componentDidMount` event, which required using a class component instead of a functional component. Now you can use the `useEffect` hook, as shown in the next exercise.

Exercise 11b.12 — FETCHING DATA IN FUNCTIONAL COMPONENT

1. Comment out the existing companies state array definition in `CompanyBrowser.js`.
2. Locate the `companies-tech.json` file in the starting files that have been provided to you as part of this lab and move it into the `public` folder.
3. Modify the function as follows:

```
import { useState, useEffect } from 'react';

const CompanyBrowser= (props) => {
   // initially the companies array will be empty
   const [companies, setCompanies]  = useState([]);

   useEffect( () => {
      console.log('useEffect');
      // Here we are using a json file in the public folder.
      // In a real-world example, you'd use the URL of a web API
      const url = '/companies-tech.json';
      if (companies.length <= 0) {
         fetch(url)
           .then( resp => resp.json() )
           .then( data => setCompanies(data))
           .catch( err => console.error(err));
      }
    },[]);
```

*The second parameter to use effect is an empty array: this **MUST** be there as it tells React to only run this effect on the initial mount (i.e., the first time the component is rendered). If you didn't include the [] then it would re-fetch every single re-render, which can easily overload a server.*

4. Test. You will now see many companies now.

   Make sure you see the useEffect console showing up once. If it is not, close the browser and fix it!!

5. If you try selecting a company, you will notice that the HQ field is always empty because the web API doesn't return a field/property named `hq`: it is named `address` instead. Modify `CompanyForm` to fix this problem.

TEST YOUR KNOWLEDGE #1

1 Open `lab11b-test01-markup-only.html` in the browser.

This provides the markup needed for the exercise.

2 Create a new React project using Vite.

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 11b.9. 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 11b.10.

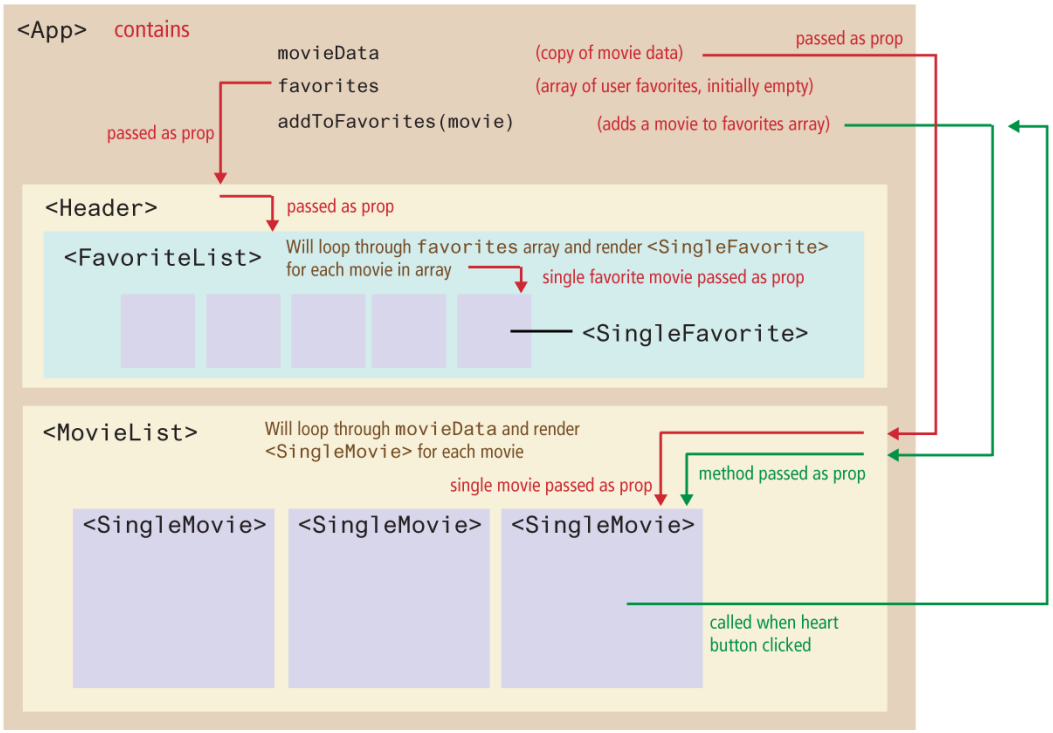

Figure 11b.9 – Data flow between components

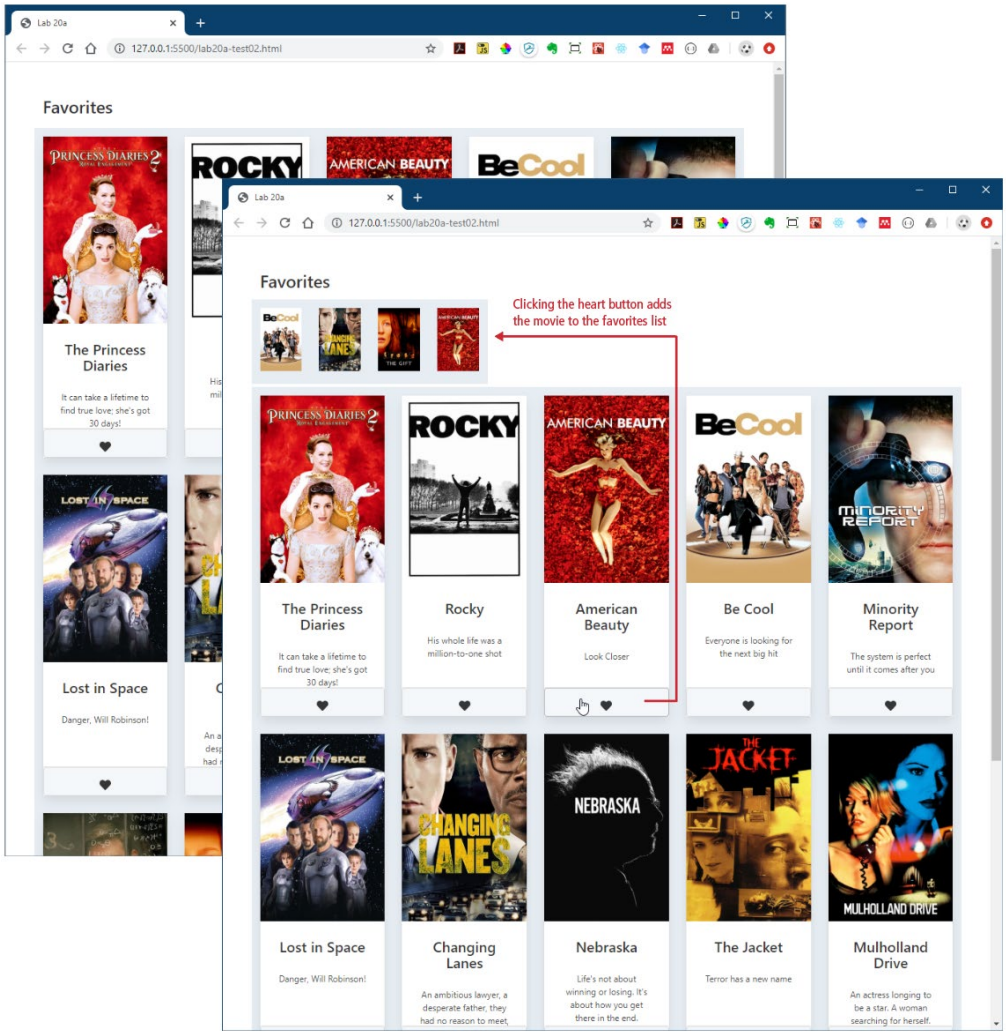

Figure 11b.10 – Finished Test Your Knowledge #2