Introduction to React

Last week we introduced React in the lectures and spent a bit of time writing some simple function-based components as part of our exploration of JSX. This time we are going to revisit some of that material, but we are going to get more serious quite quickly. As before, you may work with CodeSandbox, as it will save you quite a bit of pain in the early stages, but you may also look at David's React Setup Guide which you will find on BB. This will enable you to get started with React development on your own machine or on the Linux VMs made available in this unit.

You can always get a pre-prepared React application by going to this link, and remember that you can fork your existing work if you want to try something out and not mess with your existing code. For local development, have a look at David's guide and the use of `create-react-app`.

You should remember that React is a view creation library, one which does not have strong opinions about the retrieval, storage and manipulation of your data. We will begin with an example which uses a local data source and over time we will connect it to one available on the web.

General Instructions - Same Each Time

As you go through, you'll see some '**Exercise**s. Make sure you try these out before moving on (where there will be solutions). You won't pick up much from this if you just copy the code I have written directly. I already know React, so you putting my code directly into your application without thinking about it first doesn't help *you* learn React.

Note: this guide will be making use of the latest ECMAScript standards (ES2015 and higher). This is assumed knowledge, but I'll try my best to explain things where necessary.

Note again: This prac expects a minimum of React `16.8.0` (the current stable release at the time of writing is `16.12.0`). We'll be making use of hooks (for those that have used React before - hooks *greatly* simplify a lot of things about React and make it easier for others to pick up).

Last Week - JSX and Your First React App

Recall that JSX is the HTML-style syntax that React uses to declare elements. You used it for the first time last week and you can read a lot more about it here.

We are going to begin by taking a closer look at a basic React application. We will begin with the simple starter kit here, which you should fork as we get

started. Let's consider each of the pieces in more detail:

```
import React from 'react';
```

Even though `React` isn't explicitly used in the file, any time you see JSX being used, you *must* import `React` into your application. CodeSandbox will remind you if you forget.

```
import ReactDOM from 'react-dom';
```

`React` itself doesn't necessarily have anything special for the Web - that's how projects such as React Native are possible. `ReactDOM` contains the special functionality to connect your React application to the DOM.

```
function App() {
  // ...
}
```

This is a React component. Yup. It's just a function that returns a valid 'Node' - you can think of this as being either some JSX, or `null`. We'll cover what's inside the component in a moment.

```
const rootElement = document.getElementById('root');
```

This, again, is not React-specific - the `document` referenced there is the webpage itself. If you look inside `public/index.html`, you'll see that there is a single `div` with an id of `root`. This is where we want our React application to 'mount' (attach itself).

```
ReactDOM.render(<App />, rootElement);
```

Here is where we combine `React` with `ReactDOM` - we render the `App` component, and tell `ReactDOM` to render it to the `rootElement`. This is the root of our React application - everything else we render will come underneath this.

Alright! There is some content on the screen. Let's quickly go through what's inside `App`.

The first thing is a 'wrapping `div`'. These are common in HTML, and in React. A `div` is used simply as a container to hold your other elements. It is important to note that a React component can only return *one* element at the top level (that element can have as many *children* as you like).

```
<div className="App">
```

The above creates a `div` with a class of `App` - in React, we don't use '`class`' because `class` is a reserved keyword in JavaScript. ReactDOM will take care of adding it correctly when it gets to the browser.

Inside the `div`, we have two elements, which are just standard HTML headings.

Some things to note:

- An `element`, in React nomenclature, is a DOM-native element, like `div` or `p`. These are basically HTML tags.
- A `component` is a custom element that you (or someone else) has created in React, like `App` above.
- All `element`s start with a *lowercase* letter (as they would in HTML)
- All `component`s *must* start with an *Uppercase* letter

Last week we spent a lot of time exploring JSX and then creating a presentation component, one which took data from our list of animals and formatted it in a way which made things look a little more interesting. It was very basic, but it was a start. This time we are going to move toward components with some *state*.

Adding Some Functionality

We are now going to add a 'Like' and 'Dislike' button, and keep track of the overall number of 'Likes'. The first thing we need to do is add the buttons and a place to display the count. Buttons are added straightforwardly via the `button` tag.

Exercise: Add 2 `button`s - one that says 'Like' and one that says 'Dislike', and add a `p` that says 'Overall likes: 0' (at the moment this will be static). Update the `h1` with a relevant title (you can get rid of the `h2`).

Answer:

It doesn't matter what else is in your `App`, but it should look something like this:

```
function App() {
  return (
    <div className="App">
      <h1>Like Counter</h1>
      <p>Overall Count: 0</p>
      <button>Like</button>
      <button>Dislike</button>
    </div>
  );
}
```

You can go ahead and click on these buttons, but nothing will happen. We need to add some *state*. State is the part of your application that remembers what has happened.

To add state, we're going to be using *Hooks* - these are a relatively new development in the React ecosystem, but they are part of the stable release and greatly simplify React overall. The previous way we did this was with Class Components which added a lot of overhead to the learning/understanding process. I recommend you read about them eventually because a lot of companies will have

code that uses them, but for now it's enough to just know that Hooks aren't the *only* way of doing this. We will cover them in the Lectures for similar reasons.

To use state, we need to import `useState`. This comes from the `React` package, and is a named export, so update you import line to look like this:

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

Then, at the top of our `App`, we need to create the state.

```
function App() {
  const [count, setCount] = useState(0);
...
```

Okay. Let's break down what's happening here. `useState` returns an `Array` (note the square brackets) with two elements - the state, and a function that can be called to modify the state. We use Array Destructuring to give these items useful names. The argument to `useState` is the initial state. So, `count` is a variable that holds the number `0` (currently), and `setCount` is a function that can be used to update the count.

This will keep track of the count over time, in between renders.

To display the count, we are going to update our `p`:

```
<p>Overall Count: {count}</p>
```

Remember that the curly-braces inside the JSX are an instruction to tell JSX that this is regular JavaScript. That means that we're going to use our new `count` variable. Saving your application, you should see *no changes* to your page, because the count is still 0. Let us now make the count 'changeable'.

A `button` has an `onClick` *prop* that we can use to help here. Most DOM elements have associated events which reflect some interaction with the page. We can *listen* for these events and respond to them using *event handlers*. We saw some old-style event-handling code in the lecture, but here we are binding the `onClick` property to work with React state. Later, we will change this function to make a sensible update to the count. For now, we're just going to set it to something static.

```
<button onClick={() => setCount(10)}>Like</button>
```

What we've done here is to work with the `onClick` prop of the `button`, and passed in a function. The function calls `setCount` with a value of `10`. If you save your page now, and press 'Like', you should see a change.

Exercise: Add an `onClick` to the 'Dislike' `button` to set the count to `-1`

Notice, however, that when you click on the buttons it just updates the state to some completely new state. We need to use the old state to compute the new state.

To make our lives simpler, we're going to create two functions, `increment` and `decrement`, that will add one and remove one from the count.

Inside `App`, below your `useState(0)` call, add the following:

```
const increment = () => {
  setCount((oldCount) => oldCount + 1);
};
```

Note the use of the anonymous lambda syntax - the `() => { };` defines a function, and we will use this frequently. And the code itself may be a little confusing - we've just passed a *function* to `setCount`? Some of you may have been expecting `increment` to look like this:

```
const increment = () => {
  setCount(count + 1);
};
```

This might work, but it is not 'safe'. The reason is that React doesn't *always* update straight away. It sometimes combines a couple of updates and does them all at the same time. That means that if you pressed the 'Like' button a bunch of times before the update applied, you'd be adding to the *old* `count` every time, and would end up with only 1 more than you had before. When you pass a function to `setCount`, it will be called with the correct value of `count` when the update is run.

Tl;dr: If you are using state to update your state, *pass a function*. Otherwise, pass the value.

Exercise: Write the `decrement` function.

Once you have both functions, update the `onClick` for the 'Like' and 'Dislike' buttons:

```
<button onClick={increment}>Like</button>
<button onClick={decrement}>Dislike</button>
```

And now, when you save, you'll have a working application that keeps track of the values over time!

Exercise: Add a *Super Like* button that adds 10 to the count every time you press it. For bonus points, make sure it can only be pressed *twice*.

Reusable Components

Now that we've built a simple application, we should try and extend it. The goal for this section is for us to be able to have a bunch of headings, each with its own individual `Like` and `Dislike` buttons and counter.

The first thing we need to do is make a *new* React component for the like and dislike buttons. A component is a nice, simple unit of code that is essentially

independent from anything else. We isolate components by making a new function and including the relevant pieces within that function body.

First things first - let's make our component. As mentioned earlier, a component is just a function that returns some React code. `App` was our first component, and now we're going to make a second component called `LikeCounter`. To do this, create a new function above `App` that looks like this:

```
function LikeCounter() {
  /* our code will go in here */
}
```

If you want to race on ahead, try and extract the relevant pieces out of `App` that make up the Like and Dislike buttons. Otherwise, follow along.

The first thing we need to do is take the words and the buttons. Keep in mind that we can only return *one* element from a component, so we'll need to wrap our buttons and text in something. For now, we'll use a `div`, because it doesn't *really* change what anything looks like.

```
function LikeCounter() {
  return (
    <div>
      <p>Like Count: {count}</p>
      <button onClick={increment}>Like</button>
      <button onClick={decrement}>Dislike</button>
    </div>
  );
}
```

Some things to note. First, I've changed 'Overall Count' to 'Like Count' - keep in mind that we want to show the number of likes per title. Secondly, note that we wrap the element we've returned in parentheses. This is technically optional, but as noted last time, it is very good practice to avoid the problems that can arise from
Automatic Semicolon Insertion. You don't need to think about it too much - just remember to wrap your return statements in parentheses if they span multiple lines.

Now that we have our `LikeCounter` component, we can bring across the state. In this case, the state will reflect the individual count, and is used only in the one component - nothing else needs to reference it, so we can bring it into our new component with no problems.

```
function LikeCounter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount((oldCount) => oldCount + 1);
  };
```

```
  const decrement = () => {
    setCount((oldCount) => oldCount - 1);
  };

  return (
    <div>
      <p>Like Count: {count}</p>
      <button onClick={increment}>Like</button>
      <button onClick={decrement}>Dislike</button>
    </div>
  );
}
```

Now, inside `App`, you can remove the state, the 'Overall Count' text and the Like + Dislike buttons. Then, add in your `LikeCounter` as shown below. When you save, you should notice only a difference in the text - everything should work in the same way as before.

```
function App() {
  return (
    <div className="App">
      <h1>Like Counter</h1>
      <LikeCounter />
    </div>
  );
}
```

Cool! Now we've made a reusable likes component! You can add as many `LikeCounter`s as you like to the page, and they'll all work independently.

The next thing we need to do is make a component to hold our titles + like buttons. What we're aiming for is something that looks like this:

```
<Headline title="Hello, World!" />
```

Inside the component we will have an `h1` elementy that displays the title text, and the `LikeCounter` from above. We'll have to figure out how to access the text we've passed in.

You will recall that this is called a `prop` in React. A `prop` is something you pass into a React component. You can think about it like the parameter to a function (because they are!). You saw this last week, but if you need the revision, try the simple examples below. Every React component you create has all its `prop`s passed to it as an object in the first argument:

```
function Test(props) {
  console.log(props);
  return null; // every component *must* return something
}
```

Now, in the following examples, `props` will be an object that contains whatever values you pass in. For `String` values, this is straightforward; for others, you need to tell the system that we are working in JavaScript:

```
<Test title="Hello" />
// ‘props‘ is an object like this: { title: Hello }
<Test number={5} />
// ‘props‘ is an object like this: { number: 5 }
<Test fn={() => console.log('hello!')} enabled={true} disabled={false} />
// ‘props‘ is an object like this: { fn: Function, enabled: true, disabled: false }
```

As you can see, you can pass *any* values you like as `prop`s. You can also see that for values other than `String`, you must wrap it in curly braces - remember from above that the curly braces allow for regular JavaScript to be used inside JSX. The final thing you should know about `props` - if you have a `boolean` prop, and you want to set its value to `true`, you can just add the name of the prop. For false, you must still pass in the value explicitly. See the following:

```
<Test enabled disabled={false} />
// ‘props‘ is an object like this: { enabled: true, disabled: false }
```

Here, `enabled` is set to `true`, because we've used the prop name there directly. However, for `disabled` to be marked as `false`, we must specify it. On the whole, we would prefer that you set `true` explicitly as well.

Now, we probably have enough information to make our `Headline` component.

Exercise: Create the `Headline` component with the API from above. It should have an `h1` that displays the `title` you pass in as a `prop`, and a `LikeCounter`. Try this yourself before looking below for the solution.

Answer:

```
function Headline(props) {
  return (
    <div>
      <h1>{props.title}</h1>
      <LikeCounter />
    </div>
  );
}
```

Now, you can use your `Headline` in `App` as many times as you like with a bunch of titles, and they'll all show up. You can like and dislike each of them individually.

Our next step is to work with an array of headlines, and then ultimately to grab them from an external API.

Resources

- If you want to get ahead of the pack, I *highly* recommend watching the (free) EggHead course 'The Beginner's Guide to React' by Kent C. Dodds. It covers a lot of content, and is designed for beginners. Link here.
- The official React docs