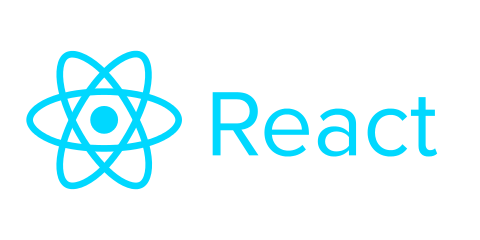
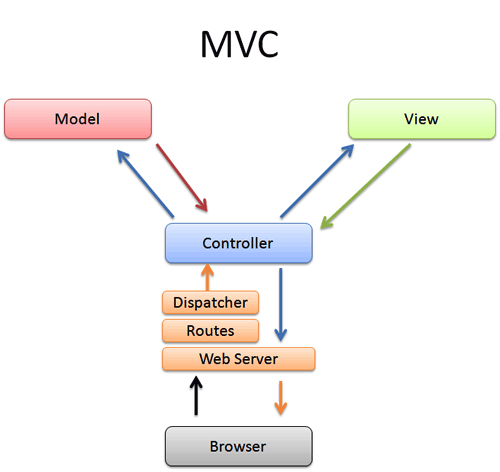
Introduction to 

React is a JavaScript library aimed for front-end development to build user interfaces. It takes care of what is displayed to the user and how they interact with our application. React is designed in a way so when the data in your app changes, the user interface will automatically update too, without the need to worry about manually updating the DOM.

Before React, when dynamic content was needed, one had to create their own rendering functions. React could be considered the View layer in the MVC concept:



An application using React, could be built from a series of reusable *components*. *Components* can represent any part of our user interface. For example, if we’re looking to create a competitor for Google Maps, you might have a Map component for viewing the map, and a LocationInput component that would allow our users to enter a location.

For easier development, React uses (or accepts) the XML format. We’re not going into the controversial historical aspects of who created XML, but suffice to point out its format:

**<tag>** contents **</tag>**

This brings us to how HTML is marked (*both XML & HTML adhere to W3C specifications*):

1. <!DOCTYPE html**>**
2. **<html** lang="en"**>**
3. **<head>**
4. **<meta** charset="UTF-8"**>**
5. **<title>**Some Title**</title>**
6. **</head>**
7. **<body>**
8. **<div** class="someClassName"**>**
9. Body Contents
10. **</div>**
11. **</body>**
12. **</html>**

We can note that every tag (like: html, head, body & div) must have a starting tag: < name > and a closing tag: </ name >. The closing tag differs from the start tag’s declaration by including a forward slash right after the starting less-than ` </ `.

In Web-Development, any HTML DOM element can either be declared directly in the HTML document, or created through JavaScript:

1. **var** ele = document.createElement("div");
2. ele.className = "someClassName";
3. ele.innerText = "Body Contents";
5. document.getElementsByTagName("body")[0].append( ele );

As we can see in the above example, we are creating an element, then we assigned its class name, given it some contents and finally appended it on the DOM. It is evident that it needs more code to create the same element result through JavaScript, and that’s why React gave us the ability to use the same syntax:

1. **function** Element() {
2. **return** (
3. <div className="someClassName">
4. Body Contents
5. </div>
6. )
7. }

**The DOM**

The DOM (or, Document Object Model) is kinda abstract, because it is what the browser use internally to structure our web-applications. The DOM isn’t just the HTML, and to get a good visual representation of what the DOM looks like, we need to open up the browser’s Dev Tools and look for the Elements tab. That visual structure (which closely resembles the HTML) is a fairly good representation of a DOM. React creates its own DOM!

To allow our apps to render efficiently, React uses a technology called the *virtual DOM*. The *virtual DOM* stores a copy of our current DOM in memory. When our app's data changes, it calculates which parts of the DOM need to be refreshed (re-rendered / repainted / altered), and only updates these elements. This makes UI built with React, updates very quickly, even for large, complex apps.

Before we start to get to know React, it is important to understand a couple of prerequisites (dependencies) that React needs in order to function properly. In JavaScript, the greater-than ( < > ) and smaller-than symbols used in XML are only acceptable in conditions.

So how React doesn’t collide with JavaScript when it is in fact JavaScript in itself?! This question brings us to explore a library called Babel. The Babel “*transpiler*” (*aka transcompiler, takes code written in a certain language, or version of a language, and converts it into valid code in a different language*) takes any XML and outputs it in plain JavaScript syntax. To better understand this in action, we need to visit: <https://babeljs.io/repl>

Babel is also responsible for future JavaScript versions (as per ECMA standards) that aren’t yet standard in all major browser vendors, and outputs as plain JavaScript in default version (ES5).

Up until React version 0.13 (aka 13), the Babel “transpiler” was included in the list of scripts needed within the <head> part of the HTML document:

1. ...
2. **<head>**
3. **<meta** charset="UTF-8"**>**
4. **<title>**Some Title**</title>**
5. **<script** src="http://fb.me/JSXTransformer-0.11.2.js"**></script>**
6. **</head>**
7. ...

With the help of the above JSXTransformer, our JavaScript with XML becomes acceptable. You can also come across files ending `\*.jsx` instead of `\*.js` to help certain environments differentiate how to deal with such files.

The other prerequisite for React is Webpack. Webpack is a very vast and complex library, that helps development with React a breeze. Webpack nowadays became quite difficult to categorize under one category, like bundler, because due to the open source community, many plugins were created for it, which also became staple, that now Webpack (with Babel plugins amongst the plethora available) gives us a build system.

**Anatomy of a Build System**

Front-end development has become more and more sophisticated, with an ecosystem of tools which has grown to allow us to process our code and assets before running or deploying them. For example, there are tools that allow us to:

1. Translate ES6 JavaScript to ES5 so it can work on older browsers
2. Shrink the size of our JavaScript and CSS code so they are served to users more quickly
3. Automatically compress images to save bandwidth
4. Add new features and syntax to make CSS easier to write

With so many tools available we have a new problem. Although we can run the tools manually this quickly becomes a burden, especially when we need to rerun the tools every time we change some code.

Build systems are pieces of software designed to help with this coordination; they are responsible for building our code so that it is ready to be run or deployed.

Loqus has its preferred build system which is based on NPM

**NPM as a build system**

The build system is based on a series of NPM scripts, each of which performs a certain task needed to build your code. The scripts live in the scripts section of the package.json file. For example, a package.json file might have the following scripts section:

...

"scripts": {

"build": "webpack",

"test": "mocha"

}

...

**Babel**

As mentioned above, Babel is the JavaScript transpiler used by our build system. For our React applications, it will play two roles:

1. Taking code written using ES6 syntax, and converting it into code that is compatible with ES5
2. Convert React’s JSX syntax into plain JavaScript code

**Webpack**

As briefly described above, we’ll be using Webpack as our module bundler. It will take our code split across multiple files and bundles it into a single file.

Webpack uses either ES6 modules (using import and export) or *CommonJS* modules (using module.exports and require) to work out which files need to be included in the bundle. We specify a single file as an entry point. Any files which are imports by our entry point are included in the bundle. Then any files which are imported by the newly added files are also included, and this continues recursively until all of our code is packaged up in a single file.

Webpack is also responsible for coordinating the processing of the files. This means that it is webpack's job to run the JavaScript files which it finds through Babel before adding them to the bundle.

**Primer on ES6**

Most developers find writing React code in ES6 a nicer experience, and most of the documentation around React takes advantage of ES6.

We’ll be going over some ES6 features that are commonly used in React apps. Specifically, we’ll cover:

1. how this works with arrow functions
2. object destructuring
3. property value shorthand
4. the spread operator
5. Object.assign
6. modules, imports and exports
7. classes

For more in-depth coverage of ES6, please visit this [set of articles](https://hacks.mozilla.org/category/es6-in-depth/).

For each of the following language features, we provide examples of usage and a description of the use cases for the feature.

**Lexical** this **with arrow functions**

If you’ve done any client-side programming in ES5, you’ve probably dealt with bugs in your code caused by this acting in counterintuitive ways. With ES6’s [arrow functions](https://strongloop.com/strongblog/an-introduction-to-javascript-es6-arrow-functions/), this behaves more consistently, and on top of that, we get a more succinct syntax for declaring functions:

1. **const** fakeTitles = [
2. 'Pirate Of Reality',
3. 'Guardians Of Hell',
4. 'Witches With Vigor',
5. 'Spies And Heroes',
6. 'Robots And Kings',
7. ];
9. **const** abbreviations = fakeTitles.map(
10. title => title.toLowerCase().slice(0, 3));
12. // equivalent using non-arrow functions
13. **const** abbreviationsEs5 = fakeTitles.map(**function**(title) { **return** title.toLowerCase().slice(0, 3); });
15. console.log(abbreviations); // ['pir', 'gua', 'wit', 'spi', 'rob']

Note how the function keyword is omitted.

The return command is *sometimes* omitted with arrow functions. For single line function bodies like the one above, return statements aren’t required.

For multi-line function bodies, {...} curly brackets are used, and return is required (if the function is meant to explicitly return a value)

1. **const** fakeTitles = [
2. 'Pirate Of Reality',
3. 'Guardians Of Hell',
4. 'Witches With Vigor',
5. 'Spies And Heroes',
6. 'Robots And Kings'
7. ];
9. **const** abbreviations = fakeTitles.map(title => {
10. console.log(title);
11. **return** title.toLowerCase().slice(0, 3);
12. });

Finally, note that the arrow function syntax can vary, depending on how many parameters/arguments the function takes:

1. **const** items = ['milk', 'bread', 'eggs', 'oranges'];
3. // when no parameters
4. items.forEach(() => console.log('another item'));
6. // when one parameter, the parentheses are optional
7. items.forEach((item) => console.log(item));
8. items.forEach(item => console.log(item));
10. // when more than one parameter
11. items.forEach((item, index) => console.log('Index ' + index + ' is: ' + item));

**Object destructuring assignment**

[Destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) is a new way of assigning values contained in Objects and Arrays, to new variables.

In this Introduction to React, we will use *object destructuring*:

1. **const** obj = {
2. a: 'apple',
3. b: 'bumblebee',
4. c: 'cat'
5. }
7. **const** {a, c} = obj;
8. console.log(a); // => apple
9. console.log(c); // => cat

This syntax is particularly valuable when importing code from one module into another, as you'll see when we discuss ES6 modules.

**Property value shorthand**

This is a convenient shorthand we can use when we need to create an object literal out of a set of variables, and we want to map variable names to keys. It is essentially the inverse of object destructuring.

1. // in es6
3. **const** x = 1;
4. **const** y = 2;
5. **const** myObj = {x, y}
6. console.log(myObj); // => { x: 1, y: 2 }
8. // equivalent to this in es5
10. **var** x = 1;
11. **var** y = 2;
12. **var** myObj = {x: x, y: y};

**Spread operator**

The [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) provides a compact way to apply parameters to functions.

1. **function** calcVolume(width, height, depth) {
2. **return** width \* height \* depth;
3. }
4. **const** values = [10, 20, 30];
5. console.log(calcVolume(...values)); // => 6000

The spread operator can also be used to copy contents of one array elsewhere:

1. **const** array1 = [1, 2, 3];
2. **const** array2 = [4, 5, 6];
3. **const** array3 = [...array1, ...array2];
4. console.log(array3); // => [1, 2, 3, 4, 5, 6]

**Object.assign**

The Object.assign method helps us manage merging Objects, and we’ll use it throughout this introduction. Properties from objects passed as the 2nd, 3rd, 4th, etc. arguments are merged into the object passed as the first argument. Properties of later arguments will overwrite properties with the same name from earlier arguments, like so:

1. **const** objA = {
2. foo: 'foo',
3. bar: 'bar'
4. };
6. **const** objB = {
7. foo: 'something else',
8. bizz: 'bizz',
9. bang: 'bang'
10. };
12. console.log(Object.assign({}, objA, objB)); // => {foo: "something else", bar: "bar", bizz: "bizz", bang: "bang"}

**Modules**

Modules allow us to split our code into multiple files, exporting objects from one file and importing them into another. Each module provides a single object for importing, which we can destructure to access the individual variables that were exported, like:

1. // file\_a.js
2. **export** **const** width = 10;
3. **export** **const** height = 5;
4. // file\_b.js
5. **import** {width, height} from './file\_a';
6. console.log(width, height);

Notice how in file\_b.js we destructure the Object exported from file\_a.js (which contains the two exported variables, width and height).

If you want to rename variables when you import them, you can add as statements:

1. // file\_b.js
2. **import** {width as tableWidth, height as tableHeight} from './file\_a';
3. console.log(tableWidth, tableHeight);

If we want to import all of the exports from a module into an object, we can use the following syntax:

1. // file\_b.js
2. **import** \* as dimensions from './file\_a';
3. console.log(dimensions.width, dimensions.height);

Each module can also export a single variable called default, which has a corresponding shorthand syntax for importing:

1. // file\_a.js
2. **export** **default** **function** area(width, height) {
3. **return** width \* height;
4. }
5. // file\_b.js
6. // Shorthand
7. **import** area from './file\_a';
8. // Long version
9. **import** {**default** as area} from './file\_a';

The default export is generally used to export the most important variable from a module. Notice how we don't need to destructure to access the default export.

**Classes**

Classes allow us to create reusable objects and methods. To declare a class we use the class keyword and add methods to the body of the class:

1. **class** Animal {
2. constructor(name) {
3. **this**.name = name;
4. }
6. speak() {
7. console.log(`${**this**.name} makes a noise`);
8. }
9. }

Classes have a constructor method that is used to set any initial properties of an object created from the class. The constructor is called when the class is *instantiated*. Instantiating a class creates a new Object that has all of the methods of the class.

To instantiate a class we use the new keyword:

1. **const** fido = **new** Animal('fido');
2. fido.speak(); // fido makes a noise
4. **const** blackie = **new** Animal('blackie');
5. blackie.speak(); // blackie makes a noise

Notice how calling the speak method for fido and blackie objects will print their own name: each is a separate instance of the Animal class.

Classes can inherit from other classes to make more specific versions of that class by using the extends keyword:

1. **class** Dog **extends** Animal {
2. constructor(name, breed) {
3. **super**(name);
4. **this**.breed = breed;
5. }
7. speak() {
8. console.log(`${**this**.name} barks`);
9. }
10. }
12. **const** lassie = **new** Dog('lassie', 'Rough collie');
13. lassie.speak(); // lassie barks

The Dog class is a more specific version of the Animal class. Note how the Dog constructor calls the super method. super calls the constructor of the parent class. In this case the parent class of Dog is Animal, so super calls the constructor of the Animal class.

The call to super is needed to delegate the initialization of the name property to the parent class. This means that we don’t need to repeat this.name = name; in both the parent and child class.

The child class ( Dog ) overrides the speak method from the parent class. We have two types of Object ( Dog and Animal ) which share an *interface* (i.e.: they both have a speak method with the same call signature), but have a different implementation. This is known as *polymorphism*.

**React Components**

React applications are made up of components. You can think of components as the building blocks of our applications.

For example, take the application you’re using to follow this course in. If you had to rewrite it using React, the top menu bar might be one component. The content of what you’re actually read now could be another component. And if any side-bars are needed/visible, that could be a 3rd component. The components would be reusable. The same body-content component would display a different course, depending whether you were studying the Introduction to React curriculum or the Advanced React curriculum.

The components themselves work by rendering a number of HTML elements. Let’s dive in code to see how this works by first cloning this repo:

git clone https://github.com/pixelwashacademy/loqus-react.git

for this application, we’re going to use a simple and basic setup to work with React. We’re loading the Babel transpiler directly in the HTML document (through the *JSXTransformer.js* module). This is going to be our boilerplate to start from. All remote files will be delivered through CDNs (Content Delivery Network).

The HTML document ( *index.html* ) contains all needed files:

* 1. We’re loading Twitter’s Bootstrap, to help us with some styling and keeping the style of our app consistent throughout any browser used.
  2. We’re loading some custom CSS style to make our example-project looks like Trello
  3. We’re loading the React library, version 15.3.2
  4. We also need React’s DOM, to render our components in the browser
  5. The JSX Transformer is loaded right after React’s library
  6. And last but not least, our JavaScript file ( *script.js* ) in which we’ll code the necessary components.

After carrying out the above cloning step, we can see 3 files (*4, including the README.md*), index.html, script.js & style.css

If the above is correct, take a good look at this setup, list any questions you might have (*to ask your mentor during the next mentor-session*) and we can now start creating some React components!

Let’s first jump into the relevant branch for this step:

git checkout components

The above step will update our script.js file with a simple stateless component, through a function, which is returning XML.

You can also visit the pull requests for comments and such, here:

<https://github.com/PixelWashAcademy/loqus-react/pull/1>

**Components**

As we saw from this first example, simple React components can be created through a JS function. This is not always possible, in cases where we need to store any data inside a component. In such cases, we’ll use the Component class from the React library. In our next example, we’re going to first “translate” our function component in a React Component. We start by changing into the next branch:

git checkout props

**Properties (aka props)**

In React, properties are known as props and these are frequently used. We can pass props to a component from a “parent” component. Like we’ve seen in the earlier example, we can use any HTML element in our React components, and, we can also use other React components in our components! A parent component can then pass any props to a child component. In this example, although the ReactDOM isn’t an actual component, it is serving as one, because there’s nothing higher than the ReactDOM in React’s hierarchy.

As we can see in the code, our function changed into a class. Like we discussed in the ES6 Primer earlier, we can extend from an existent class (being the React Component class) and then add our own. In these components we need to declare a render method, which will be automatically called by React, to render in DOM whatever we return from it.

We can see that our h2 element’s content is now different, in which we are using curly braces (aka curly brackets – { } ). These braces tell React that this should be interpreted as JavaScript. The braces are a means to “escape” JavaScript in regular text. Furthermore, we are referring to this.props which means any properties passed to this component. this.props.title will be then passed a value through the title prop, such as:

<ComponentName title=”a title” />

Any number of props can be passed to a given component, and in our current example we are passing two – title and user:

<ComponentName title=”a title” user=”user name” />

The quotation marks wrapping the value are needed because this particular value needs to be a String as we need it to be visible to the user. It can be though whatever we might need, like we’ll see later. Before we jump onto the next topic, let’s pass multiple components to the DOM, like we did in the previous example (*remember to wrap them in a* <div> *tag*):

1. <div>
2. <Trello title="Note" user="Joe" />
3. <Trello title="Comment" user="Neil" />
4. </div>

The most important fact to remember about “passing props” is to imagine a data-channel between 2 components, through which we can pass any form of data (variables, methods, etc). Usually, a parent component passes data (over props) to a child component, but we can also send from a child to a parent. Take a good look at the below example and please let us know if you have any questions:

1. // flow of data from parent to child
2. **class** Parent **extends** React.Component {
3. constructor(props) {
4. **super**(props);
6. **this**.state = {
7. person1: "John Doe",
8. person2: "Jane Doe"
9. }
10. }
12. render() {
13. **return** (
14. <div>
15. <Child toChild={**this**.state.person1} />
16. <Child toChild={**this**.state.person2} />
17. </div>
18. )
19. }
21. }

24. **class** Child **extends** React.Component {
26. render() {
27. **return** (
28. <h2>Name: {**this**.props.toChild}</h2>
29. )
30. }
32. }

The above example is a 1-way flow of data, from the Parent to the Child component. Later on we’ll see a 2-way flow in which the “child” component sends data to its “parent”. In the meantime, you can also take a look at this example on CodeSandBox:

<https://codesandbox.io/s/820x66zlz0>

**Children**

So far, when we need to use/call our components, we’ve seen the self-closing type only, like:

<Trello />

These are the most commonly used, but when we need to pass children, we need to call our components with an opening and closing tag, like:

<Trello> . . . </Trello>

children are available in props inherently, so unlike “plain” props, in which we can use the most semantic label as we wish (like, this.props.title or this.props.user), children are called and found as:

this.props.children

We can say that children are a property of props. Let’s jump into our next branch to see this:

git checkout children

In this part we can see some changes. So far we’ve passed multiple components directly to the ReactDOM.render method, wrapped in a div tag. Albeit functional, GPP (Good Programming Practices) dictates that we should create a separate component to hold in all needed components. Kind of the skeleton of our React app. This is a better practice for us humans to be able to read the code easily.

First, we can see that our component changed its h3 tag to a paragraph <p> tag. In this element we provided the necessary link to React to deal with any children passed on later, through: this.props.children

When React parses this line, it looks for anything declared between the component’s tags.

<CompName> /\* any children passed here \*/ </CompName>

In the first call, we passed a button element (*that yet isn’t doing anything*) and in the second call we passed in a bunch of text:

<Trello> <button> Button-Label </button> </Trello>

<Trello> Lorem Ipsum Text </Trello>

We also created a hook for the DOM element content so we can pass this variable to ReactDOM.

Before moving onto the next step, let’s pass another Trello component to our Layout and in the title props let’s pass HTML elements to see how it works.

<Trello title={<button className="btn-primary">Prop</button>} user="David">

children text

</Trello>

and this should get our Layout component to look like this:

1. **function** Layout(props) {
2. **return** (
3. <div>
5. <Trello title="Note" user="John">
6. <button className="btn-info">Edit / Save</button>
7. </Trello>
9. <Trello title="Comment" user="Jane">
10. children as dud-button text
11. </Trello>
13. <Trello title={<button className="btn-primary">Prop</button>} user="David">
14. children text
15. </Trello>
17. </div>
18. )
19. }

Note that now the title property’s value isn’t a String ( “John” ) but it’s starting with curly braces to prompt React that we’re passing JavaScript or JSX.

**Events**

Events in React are largely the same as the ones found in the browser’s native DOM. The only difference is how they are referred to (left native – right React):

onclick === onClick

onmouseenter === onMouseEnter

onscroll === onScroll

defaultvalue === defaultValue

you get the gist of camelCasing.

We can find a complete list of these events here:

<https://reactjs.org/docs/events.html>

So far we used some “dummy” element in place for “fake” user-interaction, like:

<h3> Edit / Save </h3>

or

<button> Some Label <button> -- although this is a button we didn’t attach any event-listener to it

Let’s start by jumping into the corresponding branch of our repo:

git checkout events

Here we can see that we’ve attached an onClick event-listener to the p tag, and when it’s triggered, it is calling the method declared within the same component named clicker.

*A note on comments in JSX*: like you might have already noticed, comments inside JSX are not the “conventional” double forward-slash ( // ) as these are a bit more laborious with an opening for the comment and another for the closing:

{ /\* and comments needed \*/ }

JSX comments are wrapped in curly braces, to inform Babel that we’re “escaping” normal declarations. Then we continue with a more “traditional” kind, with a forward-slash followed by an asterisk to open the comment-statement, and an asterisk followed by a forward-slash to close it.

**Event Handling**

After dealing with events (event listeners) now it’s time to do something with them. In the previous example, we saw how the onClick event listener was calling the clicker method. Let’s start giving some shape to our Trello-like application.

First we need to jump into the relevant branch:

git checkout event-handling

At this stage, we’re going to be able to give “life” to our applications, by having our users interact with it. Let’s do some house-cleaning of our code, and give a look to our new *script.js* file.

We can see that our Trello component received two methods, which they are being invoked on the buttons onClick listener. Although at the moment these 2 methods are simply logging out text to the console (via console.log) they will make more sense soon. When a component is loaded on the DOM, its render method is invoked automatically by React. If we need to call the render method again, to pass different elements or data, React has a very powerful architecture that helps us in doing that. Will be covering this in the next step.

Please note, that React (version 15 onwards) event-handlers, and any invoked methods, must be through a function. In earlier versions of React, we could have just said:

<element onClick={this.methodName}>

because prior to version 15, each method declared/used had to be bound to the component, but now binding is being done automatically and we must call such methods by invoking them through an ES6 arrow-function call:

<element onClick={ ()=>this.clicker() } >

Prior to version 15:

1. **class** Trello **extends** React.Component {
3. constructor() {
4. **this**.someMethod = **this**.someMethod.bind(**this**);// this declaration was needed to bind methods
5. }
6. someMethod() {
7. console.log('someMethod triggered');
8. }

11. render() {
12. **return** (
13. <div className="someClass">
14. <h2>Some Text</h2>
15. <button onClick={**this**.someMethod}>Click!</button>
16. </div>
17. )
18. }
19. }

Further information on event-handling can be found here:

<https://reactjs.org/docs/handling-events.html>

**State**

State is React’s biggest “selling point”. What is state after all?! State is a JavaScript Object, in any React.Component that we can store any needed data in it. The JS Object was chosen for its versatility because we can store any needed data-type:

1. **this**.state = {
2. name: "John Doe",
3. age: 99,
4. employed: **true**,
5. skills: ["server-side", "client-side"],
6. address: {
7. house: "Abc",
8. number: 10,
9. street: "Some Str",
10. country: "Malta"
11. }
12. };

As we can see, this.state is an Object, and like any JS Object, its key/value pairs can be whatever we need.

We already mentioned that the render method in a React.Component is automatically invoked once the component is loaded in the DOM, and, it keeps getting invoked every time there’s any change to the state. The most popular example to demonstrate this, needs a particular and very simple Component. Let’s start by switching the branch of our repo:

git checkout state

The Counter component is declaring state with a key/value of num: 0

In each of the declared methods (increase & decrease) we are changing/mutating the value of num, and since num is within our state’s “*confines*” this will trigger a call to the render method. This is the simplest example to show how re-render occurs once state changes.

The two buttons are “listening” for mouse-clicks through the onClick event, and when this is triggered, they are calling their respective methods. Both methods are changing the state, hence re-render occurs.

You can also take a look at this example on CodeSandBox, depicting re-render:

<https://codesandbox.io/s/6452pvwvvz>

**Adding State to Components**

After following the previous example (i.e.: *re-render on state change*), let’s jump back to our Trello-like application, and decide what state it needs.

First checkout the branch needed:

git checkout adding-state

In our *script.js* we can notice a drastic change from our previous version of the Trello component. Each note or comment (or trello) has 2 distinct states. In one state it is displaying the contents of note, and in the other state, it is receiving the contents of the note.

So we said that state could be either { editing: true } or { editing: false }.

We also created the respective methods that will be called to change the state, edit() & save()

The most important point to observe here, is how the render method is returning elements to the DOM. Every time the render is invoked, it first checks if this.state.editing is either true or false, and according to that condition, it then invokes its corresponding method, either renderNormal or renderForm, in which these rendering methods take care of returning/displaying the appropriate elements.

At the moment, when a user tries changing the contents of the note, it is being ignored, because we’re not yet addressing that step. How we would go about, extracting data from user inputs? Let’s get to it.

**Refs**

refs are only available within React, and such attribute is not found on native HTML elements. FaceBook created refs to give us direct access to user inputs through the same library, rather than having to go back to JavaScript for DOM manipulation. It is interesting how refs are used and declared, because within the element, we use ref (*singular*) and when we need to access that given element’s reference, we call them by this.refs

Let’s jump in, and first run:

git checkout refs

Now we can see 2 changes.

The first one is in the <textarea> element in which we’ve added the ref attribute of “newText”:

<textarea ref=”newText”>

The second is in the save() method, in which we are accessing the value inserted in the textarea:

const val = this.refs.newText.value;

For now, we are only printing ( console.log ) the text extracted from the textarea into the JavaScript Console in the browser’s Dev Tools. We are not actually saving the new text for now, because first we need to change how our application is designed! We are passing the default value of each note as children, which is not akin to the proper implementation for such an app. We need a parent component to hold the contents of each note, and pass the data to and from a child component.

**Multiple Child Components**

Instead of using a Layout skeleton for only appending our Trello components on the DOM, let’s make better use of it, and we’ll start by changing it from a stateless component ( function ) to a stateful one ( React.Component )

git checkout child-components

On this step, we have separated, and also delegated our concerns. In the Layout component we are storing each note’s contents, and depending on how many comments are in this component’s state, its respective Trello note will be mapped over. We can see on line 75 that the state is getting mapped, and through each iteration, we are creating a new Trello for that iteration’s value.

this.state.comments === [ “Note 1”, “Note 2”, “Note 3” ]

Arrays are “iterable” and the map method returns a new Array containing whatever we instruct it to. We also added a key attribute to the Trello component and this is something required by design in React. When we “feed” an “iterable” variable (like the map method generates), each generated element/component should contain a unique key. We are using the Array’s indices to pass on that unique value for each iterated element: [ 0, 1, 2 ]

If this map method looks weird, we can go over it through the more verbose ES5:

1. **this**.state.comments.map( **function**( text, i ) {
2. **return** ( <Trello key={i}>{text}</Trello> )
3. });

The map method’s callback function takes 2 arguments/parameters. The 1st argument is for the value it finds on each iteration, and the 2nd argument will be the index of that given element within the Array. In this step we used both, as we need the contents to pass as children to the Trello component, and the index for the key uniqueness.