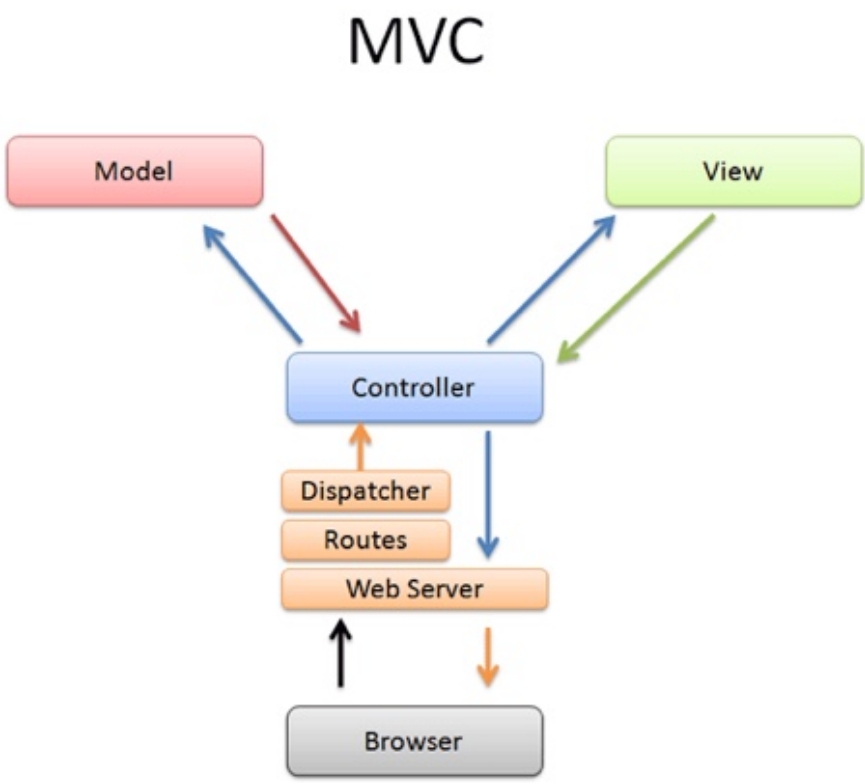
1. Introduction to React

Introduction to

React

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*):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Some Title</title>
</head>
<body>
  <div class="someClassName">
    Body Contents
  </div>
</body>
</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:

```
var ele = document.createElement("div");
ele.className = "someClassName";
ele.innerText = "Body Contents";

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:

```
function Element() {
  return (
    <div className="someClassName">
      Body Contents
    </div>
  )
}
```

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:

```
...
<head>
  <meta charset="UTF-8">
  <title>Some Title</title>
  <script src="http://fb.me/JSXTransformer-0.11.2.js"></script>
</head>
...
```

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:

- Translate ES6 JavaScript to ES5 so it can work on older browsers
- Shrink the size of our JavaScript and CSS code so they are served to users more quickly
- Automatically compress images to save bandwidth
- 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 to build our code. The scripts are declared 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:

- Taking code written using ES6 syntax, and converting it into code that is compatible with ES5
- 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:

- how `this` works with arrow functions
- object destructuring
- property value shorthand
- the spread operator
- `Object.assign`
- modules, imports and exports
- classes

For more in-depth coverage of ES6, please visit this set of articles.

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, `this` behaves more consistently, and on top of that, we get a more succinct syntax for declaring functions:

```
const fakeTitles = [
  "Pirate Of Reality",
  "Guardians of Hell",
  "Witches With Vigor",
  "Spies And Heroes",
  "Robots And Kings"
];

const abbreviations = fakeTitles.map( title => title.toLowerCase().slice(0,3) );

// equivalent using non-arrow functions
const abbreviationsEs5 = fakeTitles.map(function(title) { return
title.toLowerCase().slice(0,3) });

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)

```
const fakeTitles = [
  "Pirate Of Reality",
  "Guardians of Hell",
  "Witches With Vigor",
  "Spies And Heroes",
  "Robots And Kings"
];

const abbreviations = fakeTitles.map(title => {
  console.log(title);
  return title.toLowerCase().slice(0,3);
});
```

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

```
const items = ["milk", "bread", "eggs", "oranges"];

// when no parameters
items.forEach(() => console.log("another item"));

// when one parameter, the paratheses are optional
items.forEach((item) => console.log(item));
items.forEach(item => console.log(item));

// when more than one parameter
items.forEach((item, index) => console.log("Index " + index + " is: " + item));
```

Object destructuring assignment

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*:

```
const obj = {
  a: "apple",
  b: "bumblebee",
  c: "cat"
}

const { a, c } = obj;
console.log(a);   // => apple
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.

```
// in ES6

const x = 1;
const y = 2;
const myObj = {x, y}
console.log(myObj);   // => { x: 1, y: 2 }

// equivalent to this in ES5

var x = 1;
var y = 2;
var myObj = { x: x, y: y };
```

Spread operator

The spread operator provides a compact way to apply parameters to functions.

```
function calcVolume(width, height, depth) {
  return width * height * depth;
}
const values = [10, 20, 30];
console.log(calcVolume(...values));   // => 6000
```

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

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const array3 = [...array1, ...array2];
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:

```
const objA = {
```

```
 foo: 'foo',
 bar: 'bar'
};

const objB = {
 foo: 'something else',
 bizz: 'bizz',
 bang: 'bang'
};

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:

```
// file_a.js
export const width = 10;
export const height = 5;
```

```
// file_b.js
import {width, height} from "./file_a";
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:

```
// file_b.js
import {width as tableWidth, height as tableHeight} from "./file_a";
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:

```
// file_b.js
import * as dimensions from "./file_a";
console.log(dimensions.width, dimensions.height);
```

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

```
// file_a.js
export default function area(width, height) {
  return width * height;
}
```

```
// file_b.js
// Shorthand
import area from './file_a';
// Long version
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:

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise`);
  }
}
```

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:

```
const fido = new Animal('fido');
fido.speak(); // fido makes a noise

const blackie = new Animal('blackie');
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:

```
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  speak() {
```

```
    console.log(`${this.name} barks`);
  }
}

const lassie = new Dog('lassie', 'Rough collie');
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:

- We're loading Twitter's Bootstrap, to help us with some styling and keeping the style of our app consistent throughout any browser used.
- We're loading some custom CSS style to make our example-project looks like Trello
- We're loading the React library, version 15.3.2
- We also need React's DOM, to render our components in the browser
- The JSX Transformer is loaded right after React's library
- 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:

```
1 <ComponentName title="some 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="some 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*):

```
<div>
  <Trello title="Note" user="Joe" />
  <Trello title="Comment" user="Neil" />
</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:

```
// flow of data from parent to child
class Parent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      person1: "John Doe",
      person2: "Jane Doe"
    }
  }

  render() {
    return (
      <div>
        <Child toChild={this.state.person1} />
        <Child toChild={this.state.person2} />
      </div>
    )
  }

}


class Child extends React.Component {

  render() {
    return (
      <h2>Name: {this.props.toChild} </h2>
    )
  }

}
```

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.

```
<ComponentName>  /* any children are passed here */  </ComponentName>
```

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 this this:

```
function Layout(props) {
  return (
    <div>

      <Trello title="Note" user="John">
        <button className="btn-info">Edit / Save</button>
      </Trello>

      <Trello title="Comment" user="Jane">
        children as dud-button text
      </Trello>

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

    </div>
  )
}
```

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

```
1 <button> Some Label </button>   // although this is a button, no event-lister is attached
```

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:

```
1 <element onClick={this.methodName}>
```

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

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

In earlier versions:

```
class Trello extends React.Component {

  constructor() {
    this.someMethod = this.someMethod.bind(this); // this was needed to bind methods
  }

  someMethod() {
    console.log("someMethod triggered");
  }

  render() {
    return (
      <div className="someClass">
        <h2>Some Text</h2>
        <button onClick={this.someMethod}>Click!</button>
      </div>
    )
  }
}
```

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:

```
this.state = {
  name: "John Doe",
  age: 99,
  employed: true,
  skills: ["server-side", "client-side"],
  address: {
    house: "Abc",
    number: 10,
    street: "Some Str.",
    country: "Malta"
  }
}
```

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 `ref`erence, 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"`:

```
1 <textarea ref="newText">
```

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

```
1 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 `map`ped over. We can see on line 75 that the `state` is getting `map`ped, and through each iteration, we are creating a new `Trello` for that iteration's value.

```
1 this.state.comments     ===     [ "Note 1", "Note 2", "Note 3" ]
```

`Array`s 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 (
3     <Trello key={i}> {text} </Trello>
4   )
5 })
```

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.

Updating State and Removing Components

In the previous step, we changed our initial "*architecture*" of our app, as now the `Layout` component is responsible for each note's (`Trello`) contents, as well as where it resides in the `Array` indices. From this point onwards, our `Layout` component will keep growing, and since it isn't used anymore as a "scaffold" to prop up our components on the DOM, we should think about moving it to its own file. Let's start by doing some house-cleaning, and divide our current `script.js` into 2 smaller files: `Trello.js` and `Layout.js`. Since we are not developing our small app through a build environment (*as we'll be covering this later on in the course*), we need this change in our file/s to be reflected in our HTML document. Just like JavaScript, which is interpreted/read from left to right, top to bottom, HTML is parsed in the same way. In our `Layout` component, we are calling the `Trello` component, so the `Trello.js` file must be loaded before we load our `Layout.js` file, otherwise we'll encounter a Reference Error, that `Trello` is `undefined`!

Before we jump into the next branch, first please take a good look at the `Layout` component `render` method, Line 75, specifically the `map`'s callback:

```
this.state.comments.map(( text,i ) => ( <Trello key={i}>{text}</Trello> ))
```

Now let's checkout our branch to see this change:

```
git checkout updating-state
```

In `Layout` component we can find the `mapTrello` method, which took place in the `map` callback. If you're not familiar with JavaScript, you might find it weird that the arguments aren't passed. This is done implicitly by JavaScript, by `bind`ing the callback to the method, and any arguments are passed automatically. This example might shed more light:

```
const arr = [ "zero", "one", "two", "three" ];

// ES5
arr.map(function(val, i) {
  return (val + " is at index " + i);
});
/* outputs:
  [
    "zero is at index 0",
    "one is at index 1",
    "two is at index 2",
    "three is at index 3"
  ]
*/

// ES6
arr.map((val,i) => (`${val} is at index ${i}`))
/* outputs:
  [
    "zero is at index 0",
    "one is at index 1",
    "two is at index 2",
    "three is at index 3"
```

```
  ]
*/

// ES5 callback
function mapperES5(a, b) {
  return (a + " is at index " + b);
}

// invoking ES5 cb
arr.map(mapperES5)

// ES6 callback
const mapperES6 = (a,b) => (a + " is at index " + b);
// in the above note, that right after the "fat-arrow"
// we are using parenthesis which implicitly creates
// a `return` statement, and no need to insert `return`

// invoking ES6 cb
arr.map(mapperES6)

// both of the above produce the same exact result
// this could also be personal preference, but according
// to style guides as well as GPP, the latter is
// better for readability purposes
```

It is very important to note, that according to React's documentation and guidelines, it is not possible for React to `bind` *methods when passed as* `props`*, and in such case, we cannot rely on the implicit* `bind`*ing as it won't work. Implicit* `bind`*ing only works with absolute variables. To pass* `function`*s (i.e.: methods) we need to pass any necessary arguments. The TL;DR of this concept is that* `function`*s can make use of the* `this` *keyword and using implicit* `bind`*ing (from ES5) doesn't work, and creates unexpected behaviour. If this is unclear, or you have any questions, thoughts, doubts or concerns, please jot them all down so you'll be able to ask your lecturer during sessions.*

With the `Layout` component given more "responsibility" we should consider moving other methods to it. Let's start by taking the `remove` method. The `remove` method will be assigned to each `Trello` component, so when the user wants to delete that note, they can perfectly do so by clicking the corresponding UI/button.

As we've pointed out already, the `Layout` component is "governing" the `Trello` component/s. In order to remove a `Trello` from the `Layout`, and to update the contents of a `Trello`, we need to pass on this information/data to the `Layout` component. We can imagine that the `Layout` component needs methods that manipulate its state according to the `Trello` UI. How are we gonna make the two components communicate? We need to pass `props` from one another, and we'll see how we can send methods as `props`, as well as how a child (*i.e.:* `Trello`) can send `props` upwards to its parent (*i.e.:* `Layout`)

In order not to have too much code/changes thrown at you in one go (which helps in solidifying better your understanding of each step), we're gonna do the remove and update in the next step.

Passing Methods as Props

We need to start by changing our branch to:

```
git checkout passing-props
```

In the Layout component we've now added two new methods, the `removeNote` and the `updateNew`. As we can see, both of these methods need arguments/parameters that aren't within the `Layout` component! So far, we've only managed to pass data through `props` downstream (from parent to child) but now we need the child component to pass data to the parent component. Let's first take a look at how the `mapTrello` method is now passing the `Layout`'s methods to `Trello` through `props`:

In *`Layout.js`*, in lines 19 – 21 we are "*creating new data-channels*" (*or transmitting frequencies, if you will*) as we are passing new `props` from `Layout` to `Trello`. The newly added `props` are semantically labelled as `index`, `updateToTrello` & `removeToTrello`. This means, that when the `Trello` component "*tunes its frequency onto those channels*" it will be able to receive their data. So let's now look in *`Trello.js`* and see how it is actually accessing these new `props` "*channels*".

In this section, both files received significant changes. We are passing the methods inside `Layout` component to the `Trello` component. The `Trello` component is invoking said methods while also passing their necessary arguments. This part of React's "architecture" is one of the reasons why it garnered so much popularity with developers worldwide.

`Layout.js` received:

`removeTrello()` *which takes care of removing the passed index of the given comment in state*

`updateTrello()` *which takes care of updating the passed index & contents of the given comment in state*

`mapTrello()` *which was "refactored" with the mapping being invoked within*

`Trello.js` received:

`save()` *which now needs an index argument, and was "refactored" so it actually gets the value of the comment in* `input` *tag through* `this.refs`*, then it passes said value along with the index argument through* `props` *on* `updateToTrello` *which in return is invoking the* `updateTrello` *in* `Layout`*. Finally it flips the* `Trello`*'s* `state` *from editing* `true` *to* `false`*.*

`remove()` *which now needs an index argument, so it can pass it through* `props` *to* `removeToTrello`*, which in return invokes the* `removeTrello` *in* `Layout`*, which* `splice` *out that comment from the* `state`*'s* `Array`*.*

`renderNormal()` *now we are passing the index of the note which is available through* `this.props.index`*, to the* `Trello`*'s* `remove` *method*

`renderForm()` *now we are passing the index of the note which is available through* `this.props.index`*, to the* `Trello`*'s* `save` *method*

New Components

We're at the last step in finishing our small Trello-like application. Some steps were only taken to "illustrate" particular concepts in React. Before getting to the last features of our app, it's important to point out what are referred to as LifeCycle Methods. These lifecycle methods are built-in React, but these are going to be deprecated on the upcoming major version of this library (v17). Millions of projects and applications have been created using

such methods, and apart from the educational importance in covering them, we need to know how these work in order to be able to deal with them in our work at Loqus.

Lifecycle Methods

Let's start by going over the below table and then we'll discuss them further:

Method	When is it called?	Common Usage
`constructor(props)`	when the component is initialized	Setting the initial state
`componentWillMount()`	when the component is initialized	None; it is better to use the constructor
`componentDidMount()`	when the componnent first displays	AJAX requests, starting timers
`componentWillReceiveProps(nextProps)`	before a component receives a new set of props	Making state transitions or making AJAX requests triggered by a changing prop
`shouldComponentUpdate(nextProps, nextState)`	when a re-render is due, caused by a change in state or props	Preventing unnecessary re-renders; return false to prevent the re-render, return true to allow it to continue
`componentWillUpdate(nextProps, nextState)`	when the component will definitely re-render	Triggering animations before a re-render
`render()`	when the component needs to display something	It's mandatory!
`componentDidUpdate()`	when the component has re-rendered	Triggering third-party libraries that directly manipulate the DOM
`componentWillUnmount()`	when the component will stop displaying	Cleaning up timers and other resources associated with the component
`componentDidCatch()`	when a child of a component throws an error	Providing a clear error message to our users when something goes catastrophically wrong

These methods are extremely useful when an application needs to request data from a server. In the introduction, we pointed out that React is the View layer, of the MVC concept, and when the client-side (View) needs to request or send data to server-side, these lifecycle methods really do a big difference. Let's finish our example application, so then we can use some of the above methods and see them in action.

We need to start by checking out the next branch:

```
git checkout new-components
```

Both `Layout` and `Trello` components received new code. Let's start with `Layout`:

In our `Layout` component we can see that we flushed the `state` from `comments` and set it to an empty `Array` (*it would be weird if a user loads the application and finds dummy notes*). Then we can find a new method called `add` which takes an argument and pushes it to `state`. This method is invoked through `onClick` which we can find in a new `button` in our `render` method. It's arbitrary where to put the `String` argument, and we can leave this empty (`''`) too. The other addition to the `render` method is purely for styling purposes, in which we've added a new `div` tag, to wrap our `mapTrello` method, given a particular `className` for CSS properties. The final snippet of code that this component received, is a *lifecycle method*, which is currently *commented-out* for specific reasons. We need you to fire up the application (*there are several ways of doing this. For simplicity's sake, you can also double-click the HTML document, but if you load it through a code-editor, with Live-Server capability, you'll be able to do changes on the fly and these will be reflected in the browser once you save your code*). Running the application as is, you should see a `button` with "Add New" in it. If you *uncomment* the lifecycle method, and refresh your app, you'll see that now there's a `Trello` inside the `Layout`. This particular lifecycle, `componentDidMount` was invoked automatically by React, once the `Layout` component was appended on the DOM. In return, this method invoked the `add` method, which pushed a new `comment` to the `state`, and a `Trello` was rendered.

The only change the `Trello` component received was more or less aesthetic. If we choose to load up the `Trello` with some preset text in it (like "Edit Your Note"), it would be easier for the user to click a button, wipe out the *default* text and get focus on the `textarea`.

In the next part of this course, we'll go over 2 important aspects:

The first will be the build system used at Loqus (*so far we haven't used any, since everything is done client-side with just the JSXTransformer directly loaded in HTML*), and then we'll cover state-managers. As we could see, in React we can store data, but offloading this part, makes our applications perform better through a dedicated state-manager. Will be covering MobX to see how we can apply it in our applications.
