React Without JSX

Steven Brawer
Jan 26, 2017

introduction

This document presents a description of some features of “raw” React - that is, React without using JSX. It should be useful to a person with a little or no knowledge of React. The reader should be familiar with (but not necessarily expert in) HTML, CSS and javascript (abbreviated **js**). Most code examples are on **CodePen**, and there are links.

React has a number of functions, the most used of which is **React.createElement(…)**. (This function appears in all the examples.) JSX essentially provides a replacement for this function call, where one can use HTML-like notation instead of the function. A preprocessor turns the HTML-like element into the *createElement()* function call. Apart from this (which, nonetheless, is a big deal), everything else is vanilla js and React functions.

One problem I have in trying to learn React is that virtually all online discussions use JSX almost right from the beginning, which means there is an additional layer between js and React. I simply could not get my head around these explanations. The purpose of this document is to cut to the quick - to introduce React without using JSX. Now that I have done this, I can absorb the JSX easier.

This document really covers only the surface of React - there is much more to it than is described here. This document just creates a way for getting past the difficult lack of introductory explanatory documentation of “raw” React.

There is some stuff online. The following links were useful to me in learning.

The official React documentation, with examples, is
https://facebook.github.io/react/docs/hello-world.html .

A very useful site, which covers some of the material covered here in an exceptionally clear manner is: https://www.reactenlightenment.com/react-nodes/4.1.html .

Some sites give an overview of React which can be useful for orientation. The ones I have found are:
http://blog.andrewray.me/reactjs-for-stupid-people/
http://developer.telerik.com/featured/introduction-to-the-react-javascript-framework/
http://queue.acm.org/detail.cfm?id=2994373
https://camjackson.net/post/9-things-every-reactjs-beginner-should-know
https://medium.com/@cassiozen/10-reasons-why-every-developer-should-learn-react-87fbfef2cb91#.mos205u1u
http://betterstack.org/2016/01/12/5-reasons-why-react-js-is-awesome/

using React on CodePen

All javascript code goes in the JS window. Do settings->javascript. Then click Quick Add and make the selections in sequence: React and React DOM. In addition, add the Babel preprocessor.

You should not copy text directly from a PDF to CodePen - it adds some unrecognized characters andruins the formatting. You must export the PDF to a pure text file, and at least on the Mac, replace quotation marks in the text document with the standard quotation marks of the text editor.

Hello, World

Let's start with the traditional **hello, world** program. This is possibly the simplest React program one could write - simply print Hello, World in the browser.

Start with the following HTML:

```
<body>
        <div id = "root"></div>
        <p id="write"> display the element here</p>
</body>
```

List A

What we want to do, using js and React, is to insert a **p** element as a child of the **root** element of List A (I will refer to elements by either their id or class names, for convenience). The **write** element will display the actual p-element created by React.

To use React, I load the following(there is also a production version). Put this before the <script> section:

```
<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
```

This brings in the React libraries. Then our first, very simple javascript/React Hello, World program is:

```
<script>
var hw = React.createElement("p",{id:"what"},"Hello, World");
ReactDOM.render(hw, document.getElementById("root"));
</script>
```

List B

For the record, the entire HTML/js is:

```
<body>
<div id = "root"></div>
<p id="write"> display the element here</p>

<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>

<script>
var hw = React.createElement("p",{id:"what"},"Hello, World");
ReactDOM.render(hw, document.getElementById("root"));
</script>
</body>
```

List B1

(See *CodePen* link below. Since in CodePen, the <body> and <script> elements are not necessary, I have not indented their children.) The first function call - to **createElement ()** - "creates" a p-element *represented by* a js object. This object is pointed to by the variable **hw** but is not yet "rendered" in the browser (that is, not yet drawn in the browser window). The 2nd argument {….} is a **js object** which causes the p-element to have inline *id = "what"*. The third argument is the child of the p-element - in this case, a text string. The *createElement()* function will be discussed at length later.

We can say metaphorically that the variable *hw* "holds" the object created by *React.createElement()*. **console.log(hw)** gives this:

```
Object
$$typeof: Symbol(react.element)
_owner: null
_self: null
_source: null
_store: {validated: false}
key: null
props: {id: "what", children: "Hello, World"}
children: "Hello, World"
Object Prototype
ref: null
type: "p"
Object Prototype
```

We will be discussing the property *props* at some length later. For now we note that **console.log(hw.props)** is the js object

```
{id: "what", children: "Hello, World"}
```

Actually, the property *props* seems to have some hidden fields, but we will not need these here. The second function call in List B - **reactDOM.render()** - causes the p-element to be inserted into the DOM, which in turn causes the text *Hello, world* to be written to the browser window. The first argument is **hw**, the object representing the desired p-element, and the second argument is the parent of the p-element. The general form of this function is

```
ReactDOM.render(someElement, parentElement)
```

someElement is to be inserted as a child of **parentElement**. In this case, a new p-element is created as a child of the root element (the one with id="root", List A). The browser now looks like:

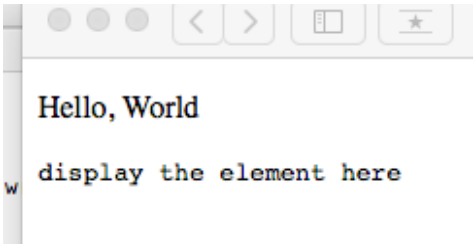


Fig AA

The p-element created is not a part of the HTML source code. Rather, it is part of the DOM hierarchy. It is an "invisible" element. But we can examine this invisible element. Add the following to the end of List B1, right under the *reactDOM.render()* statement.

```
var elem = document.getElementById("what");
document.getElementById("write").textContent=elem["outerHTML"];
```

List C

The full code is in: https://codepen.io/sbrawer/pen/YNxoyM?editors=1010 . Now the browser looks like:

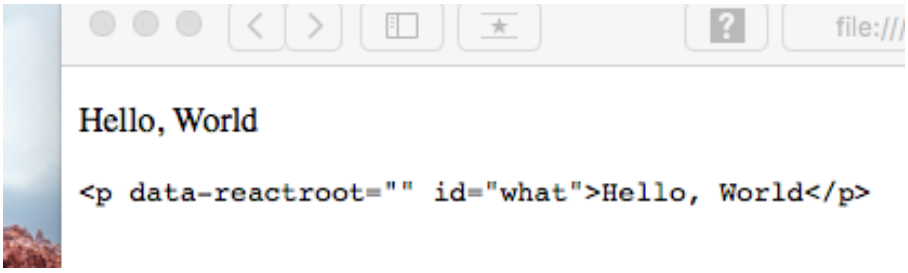


Fig BB

React has inserted the *data-reactroot* attribute, while the js object *{id: "what"}* is responsible for the *id* attribute.

There are several alternate forms for the code of List B. We could have written it this way:

```
var idObj = {id: "what"};
var txt = "Hello, World";
var rootElement = document.getElementById("root");

var hw = React.createElement("p",idObj,txt);
ReactDOM.render(hw,rootElement);
```

List D

or even this way

```
ReactDOM.render( React.createElement("p",idObj,txt),  rootElement );
```

List E

Both of these produce the same result - Fig BB.

Now we will style our Hello, World using React. We can do this from within React or using CSS. Here is the React code, without CSS.

```
var hw = React.createElement("p",{id: "what",
                                  className: "HW",
                                  style:{backgroundColor: "green",
                                         color:"blue",
                                         width: "30%"},
                                 },  "Hello, World"  );
ReactDOM.render(hw, document.getElementById("root"));
```

List F

The full code is at: https://codepen.io/sbrawer/pen/egGOeB?editors=1010 . The browser (using List C as well) looks like (note the resize behavior)

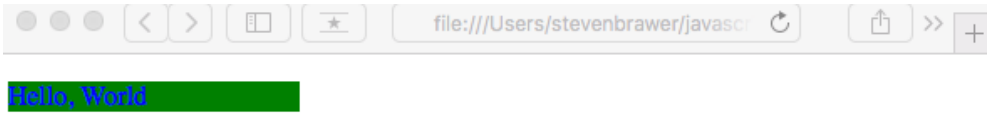


Fig CC

The 2nd argument of *createElement()* in List F is a js object. It is interpreted by React to create inline attributes in the element. React recognizes in the js object the standard HTML attribute names - in this case, **id, className, backgroundColor, style, ….** - and creates the attributes and styles accordingly. The inline styles are shown in Fig CC.

Alternatively, we could have added the following CSS to the <style> section of our HTML.

```
.HW{
        background-color: green;
        color: blue;
        width: 30%;
}
```

List G

Using this CSS style, the code of List F now becomes:

```
var hw = React.createElement("p",{id:"what", className="HW"},"Hello, World");
ReactDOM.render(hw, document.getElementById("root"));
```

List H

As you might expect by now, this gives the result:

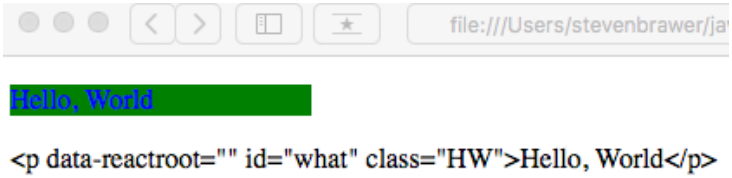


Fig DD

Note that there are **no inline styles in the p-element,** as they are all in CSS and not inserted by React. Of course the styles can be arbitrarily divided between js code and CSS. At this point, you should be able to write the code for the Fig DD web page yourself.

Note that in the javascript version, hyphens are not allowed in attribute names, and the attributes are written in camelCase. Thus, instead of *background-color* we use *backgroundColor*. Instead of *font-size*, we would use *fontSize*, and so forth. In addition, for technical reasons, instead of the word *class*, we must use *className*.

Note also, in List F, that all the attribute values - green, 30%, etc - are in quotes. This is because, obviously, js and not the browser is interpreting them. They are not in quotes in List G because the browser recognizes these keywords.

Here is yet another alternative version of the js:

```
const cssStyles = {color: "blue", backgroundColor: "green", width:"30%"};
const identifiers = {id: "what",className: "HW" };

var hw = CE("p",Object.assign(identifiers , {style:cssStyles} ),"Hello, World");
ReactDOM.render(hw, document.getElementById("root"));
```

List I

We have broken up the js object into two pieces, held by the variables **cssStyles** and **identifiers**. The js function

```
Object.assign(identifiers , {style:cssStyles} )
```

merges these into a single object - the same as in List F. The result will be Fig CC, with all styles inline.

As a matter of notation, writing **React.createElement()** can become tiresome, and also it makes for dense-looking code. But because js functions are ordinary objects, we can do the following to simplify the look of List B (CE will become very important):

```
const CE = React.createElement;
const REN = ReactDOM.render;
var hw = CE("p",{id:"what"},"Hello, World");
REN(hw, document.getElementById("root"));
```

List J

or even

```
REN(  CE("p",{id:"what"},"Hello, World"),  document.getElementById("root") );
```

This is identical in function to List B.

I will use the *CE* abbreviation (but not REN) in the remainder of these notes.

It is probably useful to point out that **JSX** is simply a shorthand way of writing the *React.createElement()* statement. Thus, if using a JSX preprocessor, the *createElement* statement, list B, becomes something like

```
<p id="what">"Hello,World"</p>
```

So in JSX this expression represents a function, and is interpreted as a functikon by the preprocessor. This of course looks a lot nicer to many people and, at least initially, might be easier to understand. Everything else is exactly the same.

As previously noted, in the *hw = createElement(…)* expression, the *createElement* function returns an object which has the property *props*. The quantity **props** is a js object which, among other things, holds the properties of the js object that is the second argument of *createElement()*. Thus, in the case of List F, **console.log(hw.props)** gives:

```
{
id: "what",
className: "HW",
style: {backgroundColor: "green", color: "blue", width: "30%"},
children: "Hello, World"
}
```

List K

It turns out that there are other properties in *props*, but for now this is sufficient.

There are two purposes of the property *props:*

- **create inline styles for an element, and**
- **pass arguments to React Components**,

as we now explore.

Hello World with a React component

In any web application, HTML elements have children, which may have children of their own. To take a simple example, without belaboring it, consider:

```
<body>
  <div>
    <ul>
        <li>.....</li>
        <li>....</li>
        ....
   </ul>
 <div>
<body>
```

List L

So *div* is a **child** of *body*, *ul* is a **child** of *div*, each *li* is a **child** of *ul* and all the *li* together are **siblings**. The key feature of React is to reproduce this hieracrchy. React can treat a number of elements as a block, either for reuse or for purposes of organization. For this, React uses the **user element**. A *user element* is one which is not an HTML element name (p, div, ul, etc). For example, *foo, listGroup, heavy* can all be tag names of user elements. *Schematically*, React allows us to rewrite the above hierarchy as,for example:

```
<ListGroup>
  <ul>
        <li>...</li>
        <li>...</li>
        ....
  <ul>
</ListGroup>

<body>
   <div>
     <ListGroup></ListGroup>
  </div>
</body>
```

List M

The idea here is that *ListGroup* (the names of user-elements must start with a capital letter) acts as a separate function (or class) and the statement **<ListGroup></ListGroup>** is essentially a function call (or the invocation of a class method, as discussed later) which "puts" its hierarchy in its place. We say that *<ListGroup></ListGroup>* is a **component**. It is a **container** for an HTML hierarchy (which could contain other components).

The first thing we need to do is to show how to use user elements in practice. So we will rerwrite the Hello World example with a user element, making the end result equivalent to, schematically:

```
<div id="root">
        <Hello></Hello>
</div>

where

[ <Hello></Hello> = <div id="what">Hello, World</div>]
```

List N

(There are a few ways to represent this, and I am not entirely consistent in this document.) Here, **Hello** is a user element. Note that **all user element names must begin with a capital letter.** The user element is a component, which is a container for HTML elements and other components.

Let's just jump in with the code. Recall our abbreviation **CE** from List J.

```
var obj =  {id:"what",
                    style:{backgroundColor:"green",color:"blue",width:"30%"} };

function Hello(props) {
        return CE("div",props, "Hello, World");
}

var hw =  CE(Hello, obj, null);
ReactDOM.render( hw,document.getElementById('root') );

var elem = document.getElementById("what");
document.getElementById("write").textContent=elem["outerHTML"];
```

List O

The result is Fig CC without the *class="HW"* attribute..

The new feature here is the **function Hello().** This function is our first example of a **component**. The rest of this document is essentially about how to use components.

This function does two things (we will get to *props* in a minute):

- It defines the HTML element (or hierarchy) which is to be rendered (in this case, a *div*), and
- It defines a **user element**, in this case **Hello**, which is the container for theHTML element (or hierarchy - see List N). *Hello* represents (or is a proxy for) the *div* element (or, eventually, a whole hierarchy of elements).

Then the **CE(Hello,…)** function invocation is envisioned as creating the **fictitous** element *Hello*, List N, which then brings along whatever elements it contains (in this case only *div*) as listed in *function Hello()*.

Note that **it is the container element (Hello) which passes the style parameters to the actual HTML element(s),** through the *props* argument**.** In List O, when the *CE(Hello,obj, null) function* is executed, that function *logically* carries out (at least) three steps:

- It copies its second argument (in this case, **obj**) into the variable *props.*
- It then invokes *function Hello(props)*, with *props* as function argument The argument *props* is an ordinary js object, at least for our current purpose.
- The function (component) returns, and then *createElement()* creates an object pointed to by *hw*, and *props* is a property of that object. This object represents an HTML element (or an HTML hierarchy).

Note that we could use any of the indirect techniques described previously. For example, to use CSS for styling, we would define in List O

```
var obj = {id: "what", className: "HW"}
```

and put the rest of the styling information in the CSS tag, as in List G.

There are several alternatives to using the *function Hello()*, and we will discuss one of them later.

a more complex Hello World component

Suppose we want to put Hello, World on two lines, with Hello on one line and World on the next line. Both have the same styling. We will need two elements - say p-elements -but **the Hello function can return only a single element.** This returned element can have as many children as desired, but there must be a single parent element to be returned.

We need to do something like what is shown schematically in the code below. There, we represent *Hello* as an element, though it really should be thought of as a **container** and not part of the hierarchy. Schematically we could think of the two-line Hello World as the following (*<Hello>* being fictitous):

```
<Hello>
   < div id=“what” class=“HELLO”>
         <p style=…> Hello</p>
         <p style=…>World</p>
  </div>
</Hello>
```

List P

The *div* element is the parent which we have to insert (along with all its children). The “container” *function Hello()* returns the single element *div* , which then brings along its two children. This is done below.

```
var CE0 = React.createElement;
var CE1 = CE0;
var CE = CE0;

function Hello(props){
         return (
                  CE0("div",{id: "what", className: "HELLO"},
                           CE1("p",props,"Hello"),
                           CE1("p",props,"World")
                  ) // end CE0
         ); // end return
} // end function

var obj = { style:{backgroundColor:”green",color:"blue",width:"30%"}};

var hw =  CE(Hello,obj , null);
ReactDOM.render( hw,document.getElementById(“root") );

var elem = document.getElementById("what");
document.getElementById("write").textContent=elem["outerHTML"];
```

List Q

https://codepen.io/sbrawer/pen/xgXKWP

Before explaining this, here is the result:

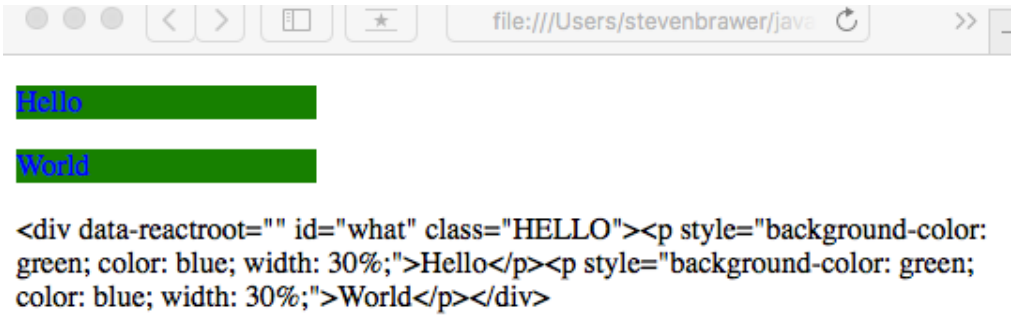


Fig EE

We can easily see the correspondence of the HTML elements in Fig EE with List P.

So what is going on in List Q? The first thing we need to explain is the general form of the *createElement* function.

```
React.createElement(tagName, anObject or null, childList or null);
```

anObject so far has been the list of properties corresponding to CSS and HTML attributes. These properties can be passed to a component via *props*. As will be seen, *props* can also contain tags which are not standard HTML tags. It also could be *null* for a plain element, with no attributes. The *childList* is a list of children elements separated by commas.

```
childList = child1, child2, child3,….
```

where all the children are siblings, and all the siblings are children of the element with *tagName*. This argument can be null if there are no children. A child can be one of three things:

- A string (such as “Hello, World”)
- A React.createElement() object
- A component

Components are a crucial part of React. For our purposes, **a component is a *createElement()* function where the tag name is a user element.** *function Hello()* is a component.

In List Q, in *function Hello()*, I have created two variables. **CE0** creates the parent element, the top level element returned by the function. **CE1** creates children of this element. **Both CE0 and CE1 are identical** - they are just written differently to indicate what is a parent and what is a child. This is useful only for reading the code - there is no functionality which depends on using CE0 or CE1.

With this explanation, it is simple to see how the *function Hello()* corresponds to the structure of List P. We see that in our case the child list is

```
childList = CE1(…), CE1(…)
```

and that this list forms the third argument of the *function CE0(..)*.

This function is invoked as a side effect of *CE(Hello, anObj, null),* where *anObj* is passed to the *function Hello()* as the *props* argument. In this case, *props* does not contain an *id* or *className* property. Rather, it contains styling attributes as properties.

time example

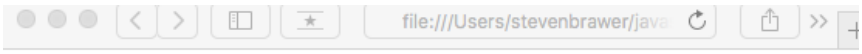
We now use what we have learned to rewrite the example given at the bottom of the web page https://facebook.github.io/react/docs/rendering-elements.html
We will do this in pure React, without JSX, The online example, which uses JSX, displays a text string with the time, and the time increments every second. Here is the React code.

```
const CE0 = React.createElement;
const CE1 = CE0, CE = CE0;

function TOCK(props){
        var str = "this is the time: " + props.tm;
        return ( CE0("div",{id: "what"},
                    CE1("h1",null,"below is the time"),
                    CE1("h2",null,str )
                )
        );
}

function tick(){
        var tme = new Date().toLocaleTimeString();
        var comp = CE(TOCK, {tm:tme},null);
        ReactDOM.render(comp, document.getElementById('root'));
}
setInterval(tick, 1000);

// wait for the clock to start
setTimeout(function(){
        var elem = document.getElementById("what");
        document.getElementById("write").textContent=elem["outerHTML"];
        }, 1500);
```

List R

https://codepen.io/sbrawer/pen/xgXRqd

Before we explain this code, here is what the result looks like.



below is the time

this is the time: 10:08:04 AM

<h1>below is the time</h1><h2>this is the time: 10:07:39 AM</h2>

Fig FF

The time on the second line increments every second (obviously this cannot be seen in Fig FF). The element *<div…>* is static, showing the time when this element was created.

The action of the program is as follows: Because of the *setInterval* function, the *function tick()* is invoked every second. When it runs, it first gets the time, storing the time as a string in the variable *tme*. The function

```
CE(TOCK, {tm:tme},null);
```

invokes the *TOCK* component (ie, invokes the *function TOCK(props)*), which creates the virtual hierarchy. The object *{tm: tme}* is copied to become part of *props*. **Note how *props* is used in this function.** In this case, *props* does not contain an HTML or CSS attribute but rather a non-HTML property **tm** whose value is the time string. Since *tm* is not standard, there would not be a *tm = ……* attribute in any element created by *function TOCK()*. In other words, if we were to have in TOCK

```
CE1("h1",props,"below is the time"),
```

the attribute tm = ….. would not appear in the *h1* element. Try it yourself. Rewrite the CE0 element in List R as

```
CE0("div",Object.assign({id: "what"}, props),…..)
```

The function TOCK(props) is a component. It is a function call (in this example) and it does what function calls do: it encapsulates a bit of functionality in a separate area, where it can be reused (or where it is logically organized). Note that TOCK is not really a functionality so much as it is the definition of an HTML section which is fixed at a certain time. In this it might more resemble a class in an object-oriented approach. In fact, in javascript, classes are functions, and unsurprisingly components can also be created using js classes, as we show later.

Since the function *tick()* is called every second, the hierarchy is created and rendered every second. As a result, the property *props* changes every second, but it is a property in an object (pointed to by *comp*) which is created anew every second..

There is one important rule for using *props*. *props* must be an **immutable** variable That is, once *props* is created in the *CE(TOCK, {tm:tme},null)* function, it should not be modified. That is, since *props* is a property of the object created by *createElement*, we **should not** do something like the following.

```
var i=0;
var comp;
function tick(){
        var tme = new Date().toLocaleTimeString();
        i++;
        if(i === 1)
                comp = CE(TOCK, {tm:tme},null);
        else
                comp.props.tm = tme;
        ReactDOM.render(comp, document.getElementById('root'));
}
```

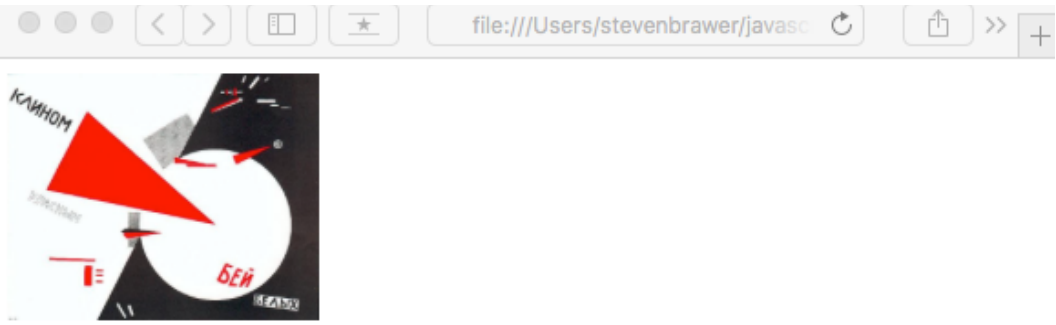
List S

This is not what happens in List R. Rather, the function *CE(TOCK, {tm:tme},null);* is called anew at each “tick”, so that a given *props* itself is not modified, but rather a new object *comp* is created on each pass, and *props* along with it.

multiple components example

In order to show how the use of components can organize code, possibly with reusable components, our next example is much more complicated. We rewrite the entire example in https://facebook.github.io/react/docs/components-and-props.html
in the section called “Extracting Components”, using pure React, without JSX. The original example uses JSX.

Here is the example, slightly modified, and written in pure React. First, here is the result.



ElLissitzky
I hope you enjoy this constructivist image!
todays date: 1/25/2017

<img class="figure" src="images/ElLissitzkyImage.jpg"
alt="ElLissitzky" width="30%" height="auto"> ElLissitzky I
hope you enjoy this constructivist image! todays date: 1/25/2017

Fig GG

One of the things to be aware of in this code is how the data displayed in the page (the name, the text, the date, the image) makes its way into the React functions. Here is the code used to create this web page.

```
const CE = React.createElement;
const CE0 = CE, CE1 = CE, CE2 = CE;
var stlGlobal = {className:"comment", id:"what"};

function formatDate(date) {
  return date.toLocaleDateString();
}

function HTMLstuff(props){
   console.log(props);
   return(
         CE0("div" ,Object.assign(stlGlobal, props.stl),    // CE1 is first child in child list of CE0
                  CE1("div",{className:"UserInfo"},       // CE2 is child of CE1
                           CE2("img",{ src:props.author.imagePath,
                                       alt:props.author.name,
                                      width: "30%", height:"auto"
                                     },null )
                  ), // CE1
                  CE1("div",null, props.author.name),      // siblings
                  CE1("div", null, props.text),
                  CE1("div",null, "todays date: " +formatDate(props.date))
         ) // CE0
    ) // return
} // end function

const comment = {
   date: new Date(),
   text: 'I hope you enjoy this constructivist image!',
   author: {
     name: 'ElLissitzky',
     imagePath: 'images/ElLissitzkyImage.jpg'
   }
};

// This creates "props", using data from variable "comment"
ReactDOM.render(
         CE(HTMLstuff,{date:comment.date,
                       text:comment.text,
                       author:comment.author,
                       stl:{style:{paddingBottom:20} }
                       } ,null), document.getElementById("root")
         );
```

```
var elem = document.getElementById("what");
document.getElementById("write").textContent=elem["outerHTML"];
```

List T

This code is found at: https://codepen.io/sbrawer/pen/apLByW .

Schematically this creates HTML something like this:

```
<HTMLstuff>
        <div id="what">
                <div class="UserInfo">
                        <img src = ….>
                </div>
                <div> a name</div>
                <div>I hope you enjoy….</div>
                <div>a date</div>
        </div>
</HTMLstuff>
```

List U

Note that the *component* **<HTMLstuff>** represents a container, not an actual element in the DOM hierarchy. The *props* created by *CE(HTMLstuff,…..)* is the following object.

```
{
date: a date,
text: "I hope you ….",
author:  {name: a name, imagePath=….a path},
stl: {style:{padding….}}
}
```

List V

In List T, the 2nd argument of *CE(HTMLstuff, {…},….)* is the link between the user data - which in this simple example is hard wired into the variable *comment,* but could come from a database - and the HTML created by React.

Based on previous discussions, the use of *props* in this code should be clear. Now we break List T into several components and make the data transmission a little more complicated.

First we pull out the *img* tag and make it a **component**, so our code becomes:

```
const CE = React.createElement;
const CE0 = CE, CE1 = CE, CE2 = CE;
var stlGlobal = {className:"comment", id:"what"};

function formatDate(date) {
  return date.toLocaleDateString();
}

function IMG(props){
   return(  CE2("img",{src: props.user.imagePath,
                              alt: props.user.name,
                              width: "30%", height: "auto"},null)
          ); // return
} //end function

function HTMLstuff(props){
    console.log(props);
    return(

          CE0("div" ,Object.assign(stlGlobal, props.stl), // CE1 is 3rd argument

                    CE1("div",{className:"UserInfo"},  // CE2 is 3rd argument
                              CE2(IMG,{user: comment.author},null)
                    ), // CE1

                    CE1("div",null, props.author.name),
                    CE1("div", null, props.text),
                    CE1("div",null, "todays date: " +formatDate(props.date))
          ) // CE0
     ) // return
} // end function

const comment = …….
ReactDOM.render(……
var elem……
```

List W

https://codepen.io/sbrawer/pen/zNEoJz?editors=1010

We now have two components: **IMG** and **HTMLstuff**. The *IMG* component is nested in the *HTMLstuff* component. Here is something very important: **the *props* passed to *IMG* is not the same as the *props* passed to *HTMLstuff*.** (I have added some logging to the CodePen code to illustrate this.)

The *IMG props* (if I can call it that) results from the function

```
CE2(IMG,{user: comment.author},null)
```

Note that *IMG props* has the key **user**, not **author**. That is, *IMG props* is the object

```
{user: {name: …., imagePath: ….}}
```

The idea is to make IMG a reusable component, not dependent on the particulars of the way the data is stored, in this case in the *comment* object. So it expects a *props* with a *user:* key. Essentially, *user* is a dummy variable. This is like having a subroutine argument name be different from the name of the variable used in passing a value to the function argument. Note the important fact that **in the function CE2(IMG,{user: comment.author},null) the object {user: ….} does not use the local *props* argument (from HTMLstuff) - it uses the *comment* global variable directly.** The *HTMLstuff props*, on the other hand, is still given by List V.

Now let's make a 2nd component out of IMG and its enclosing div, as follows:

```
....
....

function IMG(props){
  return(  CE0("img",{src:props.person.imagePath,
                      alt:props.person.name,
                      width: "30%", height:"auto"},null)
        ); // return
} //end function

function UserInfo(props){
        return(
                CE0("div",{className:"DUMMY"},
                        CE1("div",null,
                                CE2(IMG,{person:props.user},null)
                        ),
                        CE1("div",null, props.user.name)
                ) // CE0
        ) // return
} // end function

function HTMLstuff(props){
   return (
        CE0("div" ,Object.assign(stlGlobal, props.stl),// next are 3 children
                CE1(UserInfo, {user:props.author},null),
                CE1("div", null, props.text),
                CE1("div",null, "todays date: " +formatDate(props.date))
        ) // CE0
    ) // return
} // end function

const comment = …….
ReactDOM.render(……
var elem……
```

List X

https://codepen.io/sbrawer/pen/Xpepbz?editors=1010

Note how *HTMLstuff* has been simplified to a parent *div* and 3 sibling children, one of which is itself a component.

In the *UserInfo* component, List X, because the two *div* elements in the *CE1s* are siblings, we had to add a dummy *div* element in order that the function return a single element (which has children).

Note also that, for component *IMG*, we have changed the key to *person*, and in the component *UserInfo*, the key is now *user*. **It's a very good idea to make sure you understand how the data flow occurs in this example.**

an alternate way to create a component

Let *Hello* be a component. Instead of using a function, a component can be declared and defined using a js class, as follows:

```
class Hello extends React.Component{
        render(
                all the same stuff as in the function
                But use this.props instead of props.
        );

any other desired methods.

}
```

List Y

In other words, it is the same as a function except for three things:

- It is a class.
- It has a *render()* method instead of *return()*
- It uses *this.props* instead of *props*.

Try it yourself. You can find examples in the web pages given by the links at the beginning of this document.