AJL Books Ink

REACT FOR BEGINNERS

ajlBooksInk@gmail.com

Table of Contents

Chapter zero

Introduction

I am not a web developer. My professional experience is strictly java and C++ 'back-end' applications. I recently pick up two books for 'week-end' reading; a java script introduction and a React programming introduction. As I read I wrote some coding examples and share them in this paper.

Setups

NPM (Node Package Manager) setup is very easy – look at the following

> https://docs.npmjs.com/
>
> https://create-react-app.dev/docs/getting-started/
>
> https://reactjs.org/docs/create-a-new-react-app.html

Visual Studio setup.

> Read thru the following tutorial to use Visual Studio for our React development.
>
> https://code.visualstudio.com/docs/nodejs/reactjs-tutorial

Some notes and terms.

A React **Component** is a java script class that extends the class React.Component.

A React application is built as a hierarchy of React components.

A parent can have multiple children.

In this paper we name our topmost 'parent' component **App** for all our examples.

A React component renders JSX into HTML. JSX is like HTML – more on JSX as we go along.

Term: In this paper the term **component** is a class which extends the React.Component.

Such classes are technically "**Class components**"; for now, we will use the term **component** as a shortcut.

Later in this paper will we introduce a second type of component – a **Functional component**.

Project Folders: when you create a React project there are two folders (directories) where you work in. The **public** directory and the **src** directory.

The public directory will contain your index.html file.

The src directory will contain your index.js file and your 'user defined' components.

KISS: In our examples the top-most component is always named **App**

The index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Cold Beans for lunch Ink</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

The index.js. Notice the shared id **root** with the HTML

```
import React from 'react';
import ReactDOM from 'react-dom';
import {App} from './app.js' ;

ReactDOM.render(
    <App></App>,
    document.querySelector('#root')
);
```

The App component. Notice the shared component name (**App**) with the index.js

```
import React from 'react';

export class App extends React.Component{
    …
}
```

The render function:

A **component** must extend the React.Component and implement a render function. The function must accept NO arguments and return valid JSX. The JSX will be transformed to HTML by React. JSX, as you will see, is HTML-like.

JSX uses both HTML and non-HTML tags, e.g., <Widget> . The JSX is converted to HTML **by React** each time a component is 'rendered', i.e., when the render function is invoked. React takes care of all the render calls.

Example one

First!, create you React app (see Setup section above)

The HTML.

File: index.html

Location: The project folder: /public

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hell Storms Inc.</title>
  </head>
  <body>
    <div id="containerDiv"></div>
  </body>
</html>
```

File: index.js

Location: The project folder: /public

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './app.js' ;

ReactDOM.render(
<App></App>,
document.querySelector('#containerDiv')
);
```

File: app.js

Location: The project folder: /src

```
import React from 'react';

export default class App extends React.Component {

  constructor(props){
      super(props);
      console.log("In constructor...");
   }

   render(){

   console.log("About to render...");
   var heroStyle = {
      fontSize:24,
      backgroundColor:'yellow'
   };

   return(
   <div>
      <h2>Welcome aboard</h2>
      <h3 style={{color: "red"}}>Super Heros Listing </h3>

      <p style={heroStyle}>Amerigo Vespucci</p>
      <p style={heroStyle}>Siddartha Gautama</p>
      <p style={heroStyle}>Ali</p>
      <p style={heroStyle}>Jackie Robinson</p>
      <p style={heroStyle}>Gandhi</p>
      <p style={heroStyle}>Super Man</p>

      <p>{heroStyle.backgroundColor}</p>
      <p>{heroStyle.fontSize}</p>

   </div>);
  };
};
```

Run the application: http://localhost:3000/

Check the browser's console log

Example two:

Parent provides data to the child via React.Component props java script object.

Create files: index.html and index.js as in example one.

Child component.

```
import React from 'react';
export class HeroDisplayer extends React.Component {

    constructor(props){
        super(props);
        console.log("Hero Displayer: In constructor...");
    }

    render(){

        console.log("Hero Displayer: About to render...");

        let heroStyle = this.props.heroStyle;
        let jsx =
        <div>
            <h3 style={{color: "red"}}>Super Heros Listing </h3>
            <p style={heroStyle}>Amerigo Vespucci</p>
            <p style={heroStyle}>Siddartha Gautama</p>
            <p style={heroStyle}>Ali</p>
            <p style={heroStyle}>Jackie Robinson</p>
            <p style={heroStyle}>Gandhi</p>
            <p style={heroStyle}>Super Man</p>
         </div>

      return (jsx);
    }
}
```

Parent component

```
import React from 'react';
import {HeroDisplayer} from "./heroDisplayer.js";
export default class App extends React.Component {

    constructor(props){
        super(props);
        console.log("App: In constructor...");
    }

    render(){

    console.log("App:  About to render...");

    var heroStyle = {
        fontSize:24,
        backgroundColor:'yellow'
    };

    return(
    <div>
        <h2>Welcome aboard</h2>
        <HeroDisplayer style={heroStyle}>
        </HeroDisplayer>

    </div>);
  };
};
```

SPA Application (A Single Page Application)

A Simple SPA

Install the React router library in your React project directory

```
npm i react-router-dom --save
```

We build a 'skeleton application' We jump right in with a bare bones SPA.

React Components:

In this example the corresponding HTML (after rendering App) displays the application's main menu and the application's 'home page' as the initial visible page.

The React router lib component HashRouter serves as a wrapper to the menu creation and the linking of the menu items to our application's "pages". To this end the lib provides the components Route and NavLink.

Our application has just ONE page (an **SPA**). The React routing lib logic provides the look, feel and navigation of a multi-page application. Each "page" corresponds to a user (you!) defined React component; here we create components: Home, Products, Todays Specials and Contact. For this example, these components are bare boned skeletons. One of these pages is always visible and active along with our main menu on top!

In our app.js the rendering is performed in the following order:

- App component render is called
- HashRouter component render is called
- The four NavLink renders are called
- The four NavLink renders return their JSK
- The four Route component renders are called
- The four Route component renders return their JSK
- HashRouter component render returns its JSX
- App component render returns its JSX, and the corresponding HTML (main menu and Home page) is displayed.

When a user selects a give menu item the corresponding component's render is called and returns its JSX. React converts the JSX into HTML for you. The corresponding HTML is then displayed below the **menu related** HTML.

The browser user always sees the rendered Main ***menu*** on top of the active "page". One of our components (Home, Products, Todays Specials, Contacts) is rendered below the ***menu***.

Fine print item: notice the keyword **exact** in app.js. Run the app without that keyword - then put it BACK!

The App component

```
import React from 'react';
import {Route, NavLink, HashRouter} from "react-router-dom";
import {Home} from "./home.js"
import {TodaysSpecials} from "./todaysSpecials.js"
import {Products} from "./products.js"
import {Contacts} from "./contacts.js"

export class App extends React.Component{
  render(){
     return (
     <HashRouter>
        <div>
           <h1>Welcome to Cold Beans for lunch Ink</h1>
           <p>Menu Options</p>
           <ul>
            <li><NavLink exact to= "/">Home</NavLink></li>
            <li><NavLink to="/products">Our Products</NavLink></li>
            <li><NavLink to="/todaysSpecials">Todays Specials</NavLink></li>
            <li<NavLink   to="/contacts">Contact Us</NavLink</li>
           </ul>
           <div>
            <Route exact path = "/" component= {Home}></Route>

            <Route path = "/products"           component= {Products}></Route>
            <Route path = "/todaysSpecials"  component= {TodaysSpecials}></Route>
            <Route path = "/contacts"           component= {Contacts}></Route>
           </div>
        </div>
     </HashRouter>);
  };
};
```

Our Skeleton components follow.

```
import React from 'react';
export class Home extends React.Component {
    render(){
        return (
            <div>
                <p>
                    The home page skeleton
                </p>
            </div>
        );
    }
}
```

```
import React from 'react'
export class Products extends React.Component {
    render(){
        return (
            <div>
                <p>
                    Products page skeleton
                </p>
            </div>
        );
    }
 }
```

```
import React from 'react'

export class TodaysSpecials extends React.Component {
    render(){
        return (
            <div>
                <p>
                  Todays Specials skeleton
                </p>
            </div>
        );

    }
}
```

```
import React from 'react';

export class Contacts extends React.Component {
    render(){
        return (
            <React.Fragment>
                    <ul>
                    <li>Jose Bell (201) 999-1234</li>
                    <li>Maggie Bell (201) 999-1235</li>
                    <li>Zen Pen Cruz (201) 999-1236</li>
                    <li>Malcom Pope (212) 999-1237</li>
                    </ul>
            </React.Fragment>
        );
    }
}
```

Notes:

The NavLink and Route related JSX should remind you of the following multipage related HTML.

```
<ul>
<li> <a href = “/”> Home</a> </li>
<li> <a href = “/products”>Products</a> </li>
…
</ul>
```

The Home page is the default (chosen to be rendered prior to a browser user tinkering with our main menu). This default is accomplished via the “/” link. Without the **exact** keyword all four (Home, Products, Todays Specials, Contact) “pages” will be displayed below the main menu (try removing the **exact** keywords …then put them back!).

Make Home the default page:

```
NavLink exact to= "/">Home</NavLink>
 <Route exact path= "/" component= {Home}></Route>
```

Each NavLink **to** value matches the corresponding Route **path** – this is a **MUST** for the SPA to work correctly, e.g.,

```
<NavLink to = "/products”>
<Route path = "/products" component= {Products}></Route>
```

Our SPA with some style

React allows styling which is applied to the rendered HTML. In this example we will style the main menu list <ul> and the list items <li> from our SPA example.

Play with the following CSS file to see various effects on our main menu.!

I named this file: mainMenuStyle.css and placed it in the application src directory.

```
.menu{
background-color: lightgray;
list-style-type:none;
padding: 4px;
margin:3px;
width:28%;
}

.menuItem{
    text-decoration: none;
    display: inline-block;
    padding: 8px;
}

.active{
    background-color:lightsalmon;
}
```

Now make the following changes to our SPA App component

Add an import for our CSS file:

```
import "./mainMenuStyle.css"
```

Include the two classes *menu* and *menu Item* defined in our CSS file.

```
<ul className="menu">
   <li className="menuItem"><NavLink exact to= "/">Home</NavLink></li>
   <li className="menuItem"><NavLink to= "/products">Our Products</NavLink></li>

<li className="menuItem"><NavLink to="/todaysSpecials">TodaysSpecials</NavLink
></li>

<li className="menuItem"><NavLink to = "/contacts">Contact Us</NavLink></li>
</ul>
```

Note well: we defined a CSS class called ‘active’ in our CSS file above. We did not include a reference to the class in our App component. Instead, the React routing lib uses the value to hi-light the selected item from the menu .

Note the HTML vs. JSX difference with respect to the JSX camel case keyword className. We provide a more robust SPA in later chapter.

Component State changes

We will examine state changes in a single component.

For C++ and Java programmers an object's ***state*** consists of an object's (member variable, value) pairs. Besides a '***state'*** an object normally has 'behavior' made available via the class public (or protected) methods. I will call this the **classi**c definition of state.

In a React class component things get a bit dicey since the super class React.Component has a member java script object named *state*. In React jargon state refers to this inherited *state* and NOT the classic meaning of an instance's ***state***!

At the risk of being too 'cute', Reacts *state* is the **classic state** of the component's state object!

React, upon detecting a change in the *state* will perform a re-render (update) unless the function **shouldComponentUpdate** returns false – more on this later!

A simple example follows. The *state* consists of someone's age and weight. There are also two component variables for the person's name and height.

The name (Sammy) never changes. Sammy' age, height and weight will change!

It may (should) look strange that height is NOT part of our state. It is strange! We will change the later a bit later; but for now, we proceed.

```
import React from 'react';
export class App extends React.Component{
   constructor(props){
      super(props);
      this.state = {age : 17, weight: 98};
      this.name = 'Sammy';
      this.height = 72;
      this.renderCount=0;
      this.birthdayHandler = this.birthdayHandler.bind(this);
      this.gainPoundHandler = this.gainPoundHandler.bind(this);
      this.addInchHandler  = this.addInchHandler.bind(this);
   }

   birthdayHandler = () => {
      const  newAge = this.state.age+1;
      this.setState({age: newAge}) Run the application.  Click
   };
   gainPoundHandler = () => {
      const  newWeight = this.state.weight+1;
      console.log("pound " + this.state.weight);
      this.setState({weight: newWeight})
   };
   addInchHandler = () => {
      this.height++;
   };

   render() {
      console.log("Render count " + ++this.renderCount );
      return(
         <div>
            <div>
            <p>name: {this.name}</p>
            <p>age: { this.state.age}</p>
            <p>weight: {this.state.weight}</p>
            <p>height: {this.height}</p>
            </div>
            <button onClick={this.birthdayHandler}>Birthday</button>
            <button onClick={this.gainPoundHandler}>Gain a Pound</button>
            <button onClick={this.addInchHandler}>Add an Inch</button>
         </div>
      );
   }
}
```

Run the application.

Click a few times on the **birthday** and **weight** buttons and see the display change and the render count go up with each click.

Now, click the '**Add an Inch'** button **TWO** times and notice that nothing happens on the screen and in the console log.

Now click the **birthday** button again and notice that the age increases by one and the height by **TWO**.

For the birthday and weight clicks React, upon detecting a change in the state, triggered a re-render.

React did nothing when the (non-state) variable height changed.

The final **birthday** click triggered a re-render. We see the height change.

The 'fix' is to move the height into the state.

We keep the name (Sammy) outside the state; it NEVER changes.

```
import React from 'react';
export class App extends React.Component{
    constructor(props){
        super(props);
        this.state = {age : 17, weight: 98, height: 72};
        this.name = 'Sammy';
        this.renderCount=0;
        this.birthdayHandler  = this.birthdayHandler.bind(this);
        this.gainPoundHandler = this.gainPoundHandler.bind(this);
        this.addInchHandler   = this.addInchHandler.bind(this);
    }

    birthdayHandler = () => {
        const  newAge = this.state.age+1;
        this.setState({age: newAge})
    };
    gainPoundHandler = () => {
        const  newWeight = this.state.weight+1;
        this.setState({weight: newWeight})
    };
    addInchHandler = () => {
        const  newHeight = this.state.height+1;
        this.setState({height: newHeight})
    };

    render() {
        console.log("Render count " + ++this.renderCount );
        return(
            <div>
                <div>
                <p>name: {this.name}</p>
                <p>age: { this.state.age}</p>
                <p>weight: {this.state.weight}</p>
                <p>height: {this.state.height}</p>
                </div>
                <button onClick={this.birthdayHandler}>Birthday</button>
                <button onClick={this.gainPoundHandler}>Gain a Pound</button>
                <button onClick={this.addInchHandler}>Add an Inch</button>
            </div>
        );
    }
}
```

Parents should always talk to their children

Topics:

- Passing props
- State changes

Background: Bailey is a very lovely children of a single parent. Bailey has a curfew and a pet.

Part I: the basics. Please meet Bailey

But first, the parent!

```
import React from 'react';
import {Child} from "./child.js"

export class App extends React.Component{
   constructor(props){
      super(props);
      this.childName = "Bailey";
      this.pet= "snake";
      this.petName = "Sammy";
      this.state = {curfew: 17, age: 10};
   }

   render() {
      const jsx =
      <div>
         <h1>Welcome to our crazy home</h1>
         <h2>I am Baileys Parent</h2>

         <Child name= "Bailey" age = {this.state.age}  curfew = {this.state.curfew}
                  pet={this.pet} petName={this.petName}></Child>
      </div>
      return jsx;
   }
}
```

Notes:

The variables childName, pet and petName are plain java script variables with component scope (they can be and are referenced in the render function).

We placed curfew and age data in the component's **state** object. The state is inherited from the super component React.Component. The super component provides a setState() method to change the state (replace the state with a new one). A **state** change causes the component to "re-render", i.e., React calls the component's render function immediately (well almost! – more later) after a state change so that the newly re-rendered HTML reflects the state change. We will do this in a bit!

The JSX *curfew = {this.state.curfew}* tells React to render the child component Bailey and send it via props: curfew = 17. Our props also include age and pet name. The child constructor will receive the parent provided argument **props** as the java script object {name: "Bailey", age: 10, curfew: 17, pet: "Snake', petName: "Sammy"}. The **props** argument is a React reserved variable used to communicate **from** a parent Component **to** a child Component.

Notice the curly braces **{this.state.curfew}.** The braces instruct React to evaluate what is inside the braces; here it evaluates to the variable value, i.e., 17. Try removing the braces if you **dare**!

You may ask why some data is in the state object (e.g., age) and some is not (e.g., the pet's name). We deal with that a bit later.

The Child follows. It uses four props' attributes sent by parent.

```
import React from 'react';
export class Child extends React.Component{

  constructor(props){
    super(props);
    this.mood = "Happy";
  }

  render(){
     return (
       <div>
        <h2>Hello, my name is {this.props.name}</h2>
         <p>My curfew is {this.props.curfew} </p>
         <p>I am {this.props.age} years old </p>
         <p>My pet {this.props.pet} has the name: {this.props.petName}</p>
         <p>My mood today is {this.mood}</p>
       </div>
      );
  }
}
```

Notes:

The Child constructor calls the super React.Component constructor with our parent provided props.

Double dare: try adding the following line of code in the child constructor. Then notice the error! What do you conclude besides the fact that the child is trying to misbehave?!

```
this.props.curfew = 23;
```

Part II: birthdays

This example 'explains' why we put Bailey's age in the props versus say the pet's name. We want to change the age (celebrate a birthday!) in the parent, and we want the change to trigger a parent re-render which in turn will cause a child re-render (with the new age)!

We now add birthdays. Add and bind the new birthday function in the **parent**.

```
birthdayHandler(){
        const  newAge = this.state.age+1;
        this.setState({age: newAge});
}
```

The **setState** function replaces the value of the state's age and leaves the other state variable(s) intact – in this case curfew remains at 17. The **setState** function is provided by the super class React.Component.

You (the programmer) must **bind** component functions in the corresponding constructor.

```
this.birthdayHandler = this.birthdayHandler.bind(this);
```

Add a birthday related button to the parent's render method:

```
render() {
  const jsx =
  <div>
    <h1>Welcome to our crazy home</h1>
    <h2>I am Bailey's Parent</h2>

    <button onClick={this.birthdayHandler}>Let's celebrate Baileys Birthday</button>

     <Child name= "Bailey" age={this.state.age} curfew={this.state.curfew}
                pet={this.pet} petName={this.petName}></Child>
  </div>

  return jsx;
}
```

The birthday handler changes the parent's state which causes a 're-render' which, in turn, causes a Child 're-render'.

Notice that the parent state object also contains Bailey's curfew. Why did I include the curfew? So, you can try this: add a curfew changer!

Part III: Conditions in the render

Here, we use the java script **conditional (ternary) operator** to format Bailey's curfew time. This approach is fine if the conditional logic is not too complex.

```
render(){
 const curfew =  this.props.curfew;
 return (
   <div>
     <h2>Hello, my name is {this.props.name}</h2>
     <p>My curfew is {curfew > 12 ? curfew-12+"pm" : curfew+"am"} </p>
     <p>I am {this.props.age} years old </p>
     <p>My pet {this.props.pet} has the name: {this.props.petName}</p>
     <p>My mood today is {this.mood}</p>
   </div>
```

```
  );
}
```

Try this: change the above code and compose a separate function (a curfew formatter) to do the formatting and then render:

```
<p>My curfew is {curfewFormatter(curfew)} </p>
```

This approach should be used for more complex conditional logic! Do you agree?

Part IV: mood swings

Originally (see above) our child component had no state.

A mood swing is a change in the child's mood that we want displayed. We therefore move the mood to the child's state!

```
Import React from 'react';

export class Child extends React.Component{
   constructor(props){
      super(props);
      this.state={mood: "Happy"};
      this.moodChangeHandler = this.moodChangeHandler.bind(this);
   }

   moodChangeHandler(event){
      const newMood = event.target.value;
      this.setState({mood: newMood});
   }

   render(){
      const curfew =  this.props.curfew;
      return (
      <div>
           <h2>Hello, my name is {this.props.name}</h2>
           <p>My curfew is {curfew > 12 ? curfew-12+"pm" : curfew+"am"} </p>
           <p>I am {this.props.age} years old </p>
           <p>My pet {this.props.pet} has the name: {this.props.petName}</p>
           <p>My mood today is {this.state.mood}</p>
           Mood change: <input type="text" onChange={this.moodChangeHandler}/>
       </div>  );
   }
}
```

Notes:

We introduced a simple **text input** to changes the child's mood. The onChange (try onBlur) handler assigns our Child variable mood (now part of our state) with the user provided input text, i.e., the new mood.

Run the application. Try to change Bailey's mood to **"**Super Happy**".** The console log entry should be : *Old mood: Happy changed to: Super Happy*.

Notice: React JSX event related keywords: onBlur, onChange, onClick, etc. are camel case (vs. DOM conventions).

Term: a **Synthetic Event** is a React wrapper of a DOM event. It is (better be!) browser independent. In our Child component the function moodChangeHandler's input parameter is an example of a **Synthetic Event.**

Component Class Life cycle

React does a bunch of work for us 'under the hood'. For example, invoking a component's constructor and invoking the render function.

We now look at the order of execution in our App (parent) and Child (Bailey) to see the sequence of this under-cover work relative to our (the programmer's) work.

We add some simple log messages strategically placed.

We will also take advantage of a react built-in feature. The super React.Component provides a bunch of function signatures related to a component's life cycle. If the programmer (you!) implements these functions (or a subset of them) React will invoke them for you at run-time. The call sequence of these functions is clearly indicated by the function's name.

Before we continue we will introduce/review some React stuff!

- React applications consist of components in a parent –> child hierarchy where a parent can have multiple children
- The top-most component in the hierarchy we name App (this is our convention thru out this paper). See **Note** just below!
- As a component 'comes to life' it is constructed, class variables are initialized, and its render method is invoked. After this initial render completes the component is said to be "***mounted***".
- When React detects a state change to the component it calls the component's render method again. When this 're-render" occurs, the component is said to be "***updated***".
- A component is "***mounted***" one time and may be "***updated***" numerous times.
- The render function must create and return valid JSX which React converts to HTML

Note: Recall that React knows that our App serves as the top-most component via the content of the two files

- /public/index.html
- /src/index.js files

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Cold Beans for lunch Ink</title>
  </head>

 <body>
    <div id = "root"></div>
  </body>
</html>
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import {App} from './app.js' ;

ReactDOM.render(
    <App></App>,
    document.querySelector('#root')
)
```

Let us begin some coding changes to our Bailey application. Add a new function called logger to the App component.

In the constructor add:

```
this.logger = this.logger.bind(this);
this.renderCount = 0;
this.logger("Parent constructed. Render count: " + this.renderCount)
```

Compose the function:

```
logger(logEntry){
    console.log("Lifecycle: " + logEntry)
}
```

In the parent render add a log call and send a **reference** to our logger function to Bailey.

```
render() {
   this.logger("Parent rendering. Render count: " + ++this.renderCount)
   const jsx =
   <div>
     <h1>Welcome to our crazy home</h1>
     <h2>I am Bailey's Parent</h2>
     <button onClick={this.birthdayHandler}>Let's celebrate Baileys Birthday</button>

     <Child name= "Bailey" age = {this.state.age}  curfew = {this.state.curfew}
           pet={this.pet} petName={this.petName} logger={this.logger}></Child>
   </div>

    return jsx;
 }
```

Run the application and celebrate some birthdays and input some mood swings!

Notice that mood swings change the Child (***update***) BUT does not cause a parent (App) update.

The parent does not 'know' about the mood change!

We continue; add the following to the Child component:

In the Child constructor add at the bottom

```
this.props.logger(this.props.name + " Constructed. Render count: " + this.renderCount)
```

Add to the very top of the render function:

```
this.props.logger(this.props.name + " rendering. Render count: " + ++this.renderCount)
```

Run the application and celebrate some birthdays and input some mood swings! Notice that mood swings cause a bunch of Child ***updates.*** To reduce the number of updates try replacing ***onChange*** in the mood change input tag to ***onBlur*** (note: 'blur' is the loss of focus event).

Now we will insert some of the React provide life-cycle functions to get a better understanding of our application's life cycle.

Add these two functions to the App component. Do not bind them in the constructor since they are already 'built-in'. Then, run the application and review the console log.

The App component

```
componentDidUpdate(prevProps, prevState){
  this.logger( "Parent did UPDATE.  State -> age change from: " + prevState.age + " to:
       " + this.state.age);
  }

componentDidMount(){
     this.logger("Parent did MOUNT.  State -> age: " + this.state.age);
}
```

Finally, add these two functions to the Child component (do not bind them in the constructor). Run the application and review the console log.

```
componentDidUpdate(prevProps, prevState){
    this.props.logger(this.props.name  + " did UPDATE." +
   "FROM age: " + prevProps.age + " Mood: " + prevState.mood +
    " TO: age: " + this.props.age + " Mood: " + this.state.mood);
}

componentDidMount(){
  this.props.logger(this.props.name  + " did MOUNT.  age: " + this.props.age + " Mood: "
        + this.state.mood );
}
```

Bailey revisited: a component hierarchy change

Bailey's Pet

Our earlier version of Bailey had a two-level component hierarchy. We will now add a third level by separating our Pet into its own component.

App -> Child -> Pet

The App component still determines and provides Bailey's pet and the pet's name. The App does not render the Pet; this job is delegated to Bailey. The App cannot send the Pet the pet related props directly; the App must go through the Child – the Pet's renderer, i.e., in the hierarchy order top to down.

In the App we separate Child (Bailey) related data vs. the pet related data by creating two java script objects.

The Pet has NO state and simply renders its prop data. The Pet follows.

```
Import React from 'react';

export class Pet extends React.Component{

   constructor(props){
      super(props);
      this.renderCount = 0;
   }

   render(){
      this.props.logger("Pet rendered  Render count: " + ++this.renderCount)
      let pet = this.props.petInfo.pet;
      let name = this.props.petInfo.name;

      return (
         <div>
            <p>Hello, I am a Pet {pet} and my name is {name}</p>
         </div>
      );
   }
}
```

The Child renders a **Pet** with the required props values. The Child follows.

```
import React from 'react';
import {Pet} from "./pet.js"

export class Child extends React.Component{

   constructor(props){
      super(props);
      this.state={mood: "Happy"};
      this.moodChangeHandler = this.moodChangeHandler.bind(this);
      this.renderCount = 0;
   }

   moodChangeHandler(event){
      const newMood = event.target.value;
      this.setState({mood: newMood});
   }

   render(){
      this.props.logger(this.props.childInfo.name +
        " rendering. Render count: " + ++this.renderCount)

      const curfew =  this.props.childInfo.curfew;
      return (
      <div>
          <h2>Hello, my name is {this.props.childInfo.name}</h2>
          <p>My curfew is {curfew > 12 ? curfew-12+"pm" : curfew+"am"} </p>
          <p>I am {this.props.childInfo.age} years old </p>
          <p>My mood today is {this.state.mood}</p>
          <Pet petInfo={this.props.petInfo} logger={this.props.logger}></Pet>
          Mood change: <input type="text" onChange={this.moodChangeHandler}/>
     </div> );
   }
}
```

The App (parent).

```
import React from 'react';
import {Child} from "./child.js"

export class App extends React.Component{
   constructor(props){
      super(props);

      this.childName = "Bailey";
      this.state = {curfew: 17, age: 10};
      this.petInfo = {pet: 'snake', name: 'Sammy'};
      this.logger = this.logger.bind(this);
      this.renderCount = 0;
      this.birthdayHandler = this.birthdayHandler.bind(this);
   }

   birthdayHandler(){
      const  newAge = this.state.age+1;
      this.setState({age: newAge});
   }

   logger(logEntry){
       console.log("Lifecycle: " + logEntry)
   }

  render() {
     this.logger("Parent rendering. Render count: " + ++this.renderCount)

     const childInfo={name:'Bailey', age:this.state.age, curfew:this.state.curfew};

     const jsx =
     <div>
      <h1>Welcome to our crazy home</h1>
      <h2>I am Bailey's Parent</h2>
      <button onClick={this.birthdayHandler}>Let's celebrate Baileys Birthday
      </button>
      <Child  childInfo= {childInfo} petInfo= {this.petInfo} logger={this.logger}>
      </Child>
     </div>
     return jsx;
   }
}
```

Execute the application and play with mood swings and birth days. Notice that the Pet component gets rendered 'many' times but **never** changes. It has NO state and Bailey always sends the same values via the props! This example is simple, and the Pet render is not very 'expensive' which will not always be the case in the 'real world'. We now limit Sammy re-renders!

We change the Pet component by implementing the React built-in component life cycle function: **shouldComponentUpdate.** When the parent component asks a child component to re-render, the child can ignore the re-render (a.k.a. update) request, since the shouldComponentUpdate function is invoked prior to the potential re-render. Implement this method in the Pet component; do not bind it as it is built in.

```
shouldComponentUpdate(nextProps, nextState)
    return false;
}
```

Execute the application and notice via our logging that Sammy is rendered just once!

Hum! The above life cycle function does seem a bit harsh and brutish and ripe for future issues! We now tone it down a notch. Execute the application with this change. Pet should still render just once.

```
shouldComponentUpdate(nextProps, nextState){
    const upDate =  (nextProps.petInfo.pet !== this.props.petInfo.pet) ||
                    (nextProps.petInfo.name !== this.props.petInfo.name);
    return upDate;
}
```

Run the application. Then try changing the above function. Change one of the **!==** to ***===*** and see what happens. When done put the **!==** BACK!

In the Pet component. Try removing the entire constructor and the logger prop. This will slim down the component.

A 'heads up'.

We look at another component hierarchy example

> Parent -> Child -> Grandchild
>
> The Parents send a Widget and a Smidget to the Child via props.
>
> The Child only needs the Widget, and the Grandchild only needs the Smidget. It is the Child's responsibility to pass the Smidget to the Grandchild.
>
> The Child checks the props. If the Widget has not changed the Child selfishly (excuse the personification) decides not to re-render (its shouldComponentUpdate logic returns FALSE).
>
> Oh no!! What happens when the Widget does not change BUT the Smidget does -> LOGIC ERROR!

Some two-way communication

Our current hierarchy and design do not allow Bailey to communicate up the hierarchy to the parent. The design also prohibits Bailey's pet from communicating to up to Bailey (I need to go out for a walk, now master! Ruff ruff).

We make some changes.

Pet

Add a new bind to the constructor

```
constructor(props){
        super(props);
        this.renderCount = 0;
        this.messageSender = this.messageSender.bind(this);
    }
```

Add the new function. Note the new **props** attribute for sending messages

```
messageSender(evt){
            const msg = evt.target.value;
            this.props.sendMessage(msg);
    }
```

The render now returns

```
…

return (
  <div>
     <p>Hello, I am a Pet {pet} and my name is {name}</p>
     Send message to my master: <input type="text" onChange={this.messageSender}/>
  </div>  );
```

The child (Bailey)

The child now sends the Pet a function reference that allows the pet to send back message to its 'master' (to its parent component Bailey!)

```
 <Pet petInfo={this.props.petInfo}
        sendMessage={this.messageReceiver}
        logger={this.props.logger}>
</Pet>
```

Send versus receive is relative! The new child function allows the child to **receive** messages. Regarding the props attribute **name**: this is a **send** message function from the pet's point of view.

```
 messageReceiver(msg){
       this.setState({msgFromPet: msg});
   }
```

Child to parent messages.

The child no longer displays its mood changes. Instead, it sends the new mood **up** to its parent.

We remove the mood from the child state; a mood change no longer triggers a child re-render.

```
moodChangeHandler(event){
       this.mood= event.target.value;
       this.props.childInfo.sendMessage ("My mood is changing:  " + this.mood);
   }
```

The complete child component follows.

```
import React from 'react';
import {Pet} from "./pet.js"
export class Child extends React.Component{
  constructor(props){
        super(props);
        this.state={msgFromPet: null};
        this.mood = 'Happy';
        this.moodChangeHandler = this.moodChangeHandler.bind(this);
        this.messageReceiver = this.messageReceiver.bind(this);
        this.renderCount = 0;
    }

    messageReceiver(msg){
        this.setState({msgFromPet: msg});
    }

    moodChangeHandler(event){
        this.mood= event.target.value;
        this.props.childInfo.sendMessage("My mood is changing:  " + this.mood);
    }

    render(){
     this.props.logger(this.props.childInfo.name + " Render count: " + ++this.renderCount)

        const curfew =  this.props.childInfo.curfew;
        const petMsg = this.state.msgFromPet;
        const petName = this.props.petInfo.name;
        return (
        <div>
            <h2>Hello, my name is {this.props.childInfo.name}</h2>
            <p>My curfew is {curfew > 12 ? curfew-12+"pm" : curfew+"am"} </p>
            <p>I am {this.props.childInfo.age} years old </p>
            Mood change: <input type="text" onChange={this.moodChangeHandler}/>
            <p>Message from {petName}: {petMsg  ? petMsg : 'no messages'} </p>
            <p>Here is my Pet {petName}</p>
            <hr></hr>
            <Pet petInfo={this.props.petInfo}
                sendMessage={this.messageReceiver}
                logger={this.props.logger}></Pet>
        </div>  );
    }
}
```

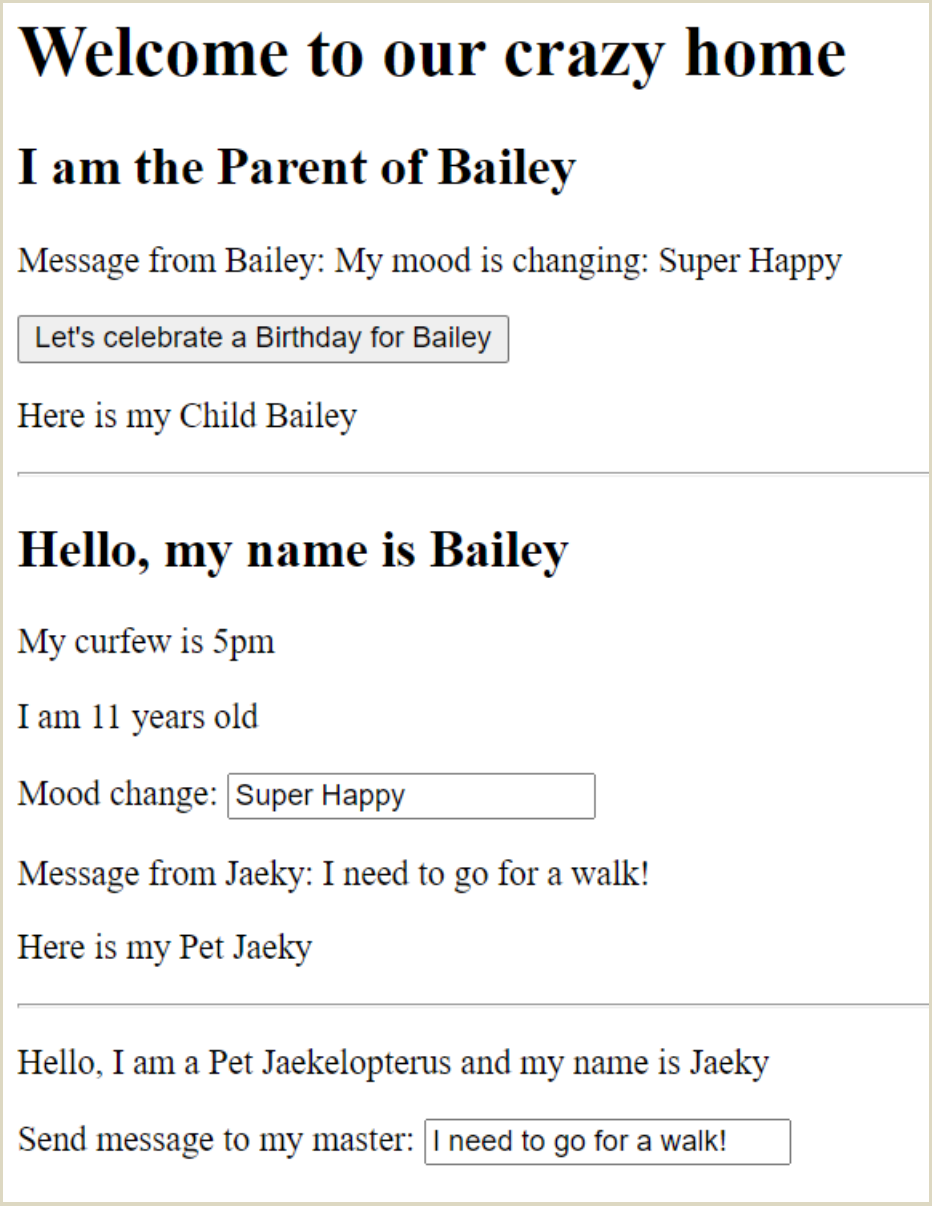
The parent

```
import React from 'react';
import {Child} from "./child.js"

export class App extends React.Component{
   constructor(props){
      super(props);

      this.childName = "Bailey";
      this.state = {curfew: 17, age: 10, childName : 'Bailey', incomingMsg : null};
      this.petInfo = {pet: 'Jaekelopterus ', name: 'Jaeky'};
      this.logger = this.logger.bind(this);
      this.renderCount = 0;

      this.birthdayHandler = this.birthdayHandler.bind(this);
      this.messageReceiver = this.messageReceiver.bind(this);
   }

   messageReceiver(msg){
      this.setState({incomingMsg: msg});
   }

   birthdayHandler(){
      const  newAge = this.state.age+1;
      this.setState({age: newAge});
   }

   logger(logEntry){
       console.log("Lifecycle: " + logEntry)
   }
```

```
   render() {
      this.logger("Parent rendering. Render count: " + this.renderCount+
       " "  + this.state.age + " " + this.state.curfew)

      const childInfo = {name: 'Bailey',
                         age: this.state.age,
                         curfew: this.state.curfew,
                         sendMessage: this.messageReceiver};

       const incomingMsg = this.state.incomingMsg;

      const jsx =
      <div>
        <h1>Welcome to our crazy home</h1>
        <h2>I am the Parent of {this.childName}</h2>
        <p>
        Message from {this.childName}: {incomingMsg  ? incomingMsg : 'no messages'}
        </p>
        <button
            onClick={this.birthdayHandler}>Let's celebrate a Birthday for {this.childName}
        </button>
        <p>Here is my Child {this.childName}</p>
        <hr></hr>
        <Child  childInfo= {childInfo}
                petInfo= {this.petInfo}
                logger={this.logger}>
        </Child>
      </div>

      return jsx;
   }
}
```

Note: we strategically placed **<hr> tags** to separate things!

Run the application and play around!



Props containing an array of data

In this example our App component sends its Cartesian Plane child an array of points. We use a plain java script class to represent one Point in the plane. Note: here we do not actually graph the points; instead, we simply list them in order.

The Point:

```
export class Point {
    // privates
    #x;
    #y;

    constructor (x, y) {
        this.x = x;
        this.y = y;
    }

    getX(){
        return this.x;
    }

    getY(){
        return this.y;
    }

    toString(){
        return "(" + this.x+ "," + this.y + ")";
    }
}
```

The Cartesian Plane.

```
import React from 'react';

export class CartesianPlane extends React.Component{
    constructor(props){
        super(props);
        this.renderShape = this.renderShape.bind(this);
    }

    renderShape(points){

        // KISS: In the exercise, instead of graphing the points we simple list them

        const jsxArr = points.map((point => <li>Point: {point.toString()}</li>));
        return jsxArr;
    }

    render(){
      return (
        <div>
          <p>Text {this.props.shape.text}</p>
          <ul>
            {this.renderShape(this.props.shape.points)}
          </ul>
        </div>
      );
    }
}
```

The parent.

```
import React from 'react';
import {CartesianPlane} from "./cartesianPlane.js"
import {Point} from "./point.js"

export class App extends React.Component{
    constructor(props){
        super(props);

        this.shape = {text: "a 5-12-13 right triangle", points:
            [
                new Point(0,0),
                new Point(5,0),
                new Point(5,12)
            ]};
    }

    render() {
        return(
            <CartesianPlane shape = {this.shape}/>
        );
    }
}
```

You should see a **complaint** about your (NOT MINE! just kidding) list items missing a unique "key" prop. This is React complaining that it wants a unique key for each list item to help it manage the JSX rendering. We fix it by adding the unique key.

```
renderShape(points)
   let I  = 0;
   const jsxArr = points.map((point => <li key = {I++}>Point: {point.toString()}</li>));
   return jsxArr;
}
```

Up until this point we have been using the term component for our classes that extend React.Component. These can be (should be!) more precisely described as a React 'class component'.

Now we take a closer look at our Cartesian Plane component. It has no state; it simply renders its props. We can make the component more lite weight by converting it from a ***Class Component*** to *a* ***Functional Component****.*

Term: a React **Functional Component** is an exported java script function that returns JSX.

Functional component

- Is light weight compared to a class component.
- Has no constructor (it is a Java Script function)
- **FOR NOW** (in this paper) our functional components will have NO state. More on this later when we look at **hook**s.
- Rendering is provided by React under the hood – like a class component

Cartesian Plane as a functional component

```
import React from 'react';

// CartesianPlane is our function name
export const CartesianPlane = (props) => {

  let renderShape = (points) => {
        let I = 0;
        const jsxArr =
          points.map((point => <li key = {I++}>Point: {point.toString()}</li>));
        return jsxArr;
    }

    return (
       <div>
          <p>Text {props.shape.text}</p>
          <ul>
            {renderShape(props.shape.points)}
          </ul>
        </div>
      );
}
```

Some related examples

The following set of examples all use the same Displayer and *'displayable' components.*

The Displayer component displays an array of *displayable components* provided by the parent component.

In this example a *displayable component* is a java script object with a toString() function!

For the *displayable component* we re-use our Pont java script object used earlier in our Cartesian Plane.

```
export class Point {
    // privates
    #x;
    #y;

    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    getX(){
        return this.x;
    }

    getY(){
        return this.y;
    }

    toString() {
        return "(" + this.x+ "," + this.y + ")";
    }
}
```

The generic displayer functional component.

```
import React from 'react';

export const Displayer = (props) => {

   console.log("Displayer is executing...");

   let buildListItem = (arr) => {|
          let I = 0;
          const jsxArr =
             arr.map((arrItem => <li key = {I++}>{arrItem.toString()}</li>));
          return jsxArr;
  }

  return (
          <div>
            <ul>
              {buildListItem(props.arr)}
            </ul>
          </div> );
}
```

Example one:

The parent.

```
import React from 'react';
import {Point} from './point.js';
import {Displayer} from './displayer.js';
export class App extends React.Component{

constructor(props){
                super(props);
                this.addedOrigin = false;
                this.state = { arr: [ new Point(2,3), new Point (4,-9), new Point(4,0) ] };
                this.addOriginPoint = this.addOriginPoint.bind(this);
        }

addOriginPoint(){
                if (this.addedOrigin === false){  // just once!
                  this.state.arr.push(new Point(0,0));

          console.log("Parent add origin. The number of points = " +
              this.state.arr.length);

                  this.addedOrigin = true;
                } else {
                  console.log("Parent add origin. Do nothing!");
                }
        }

        render() {
          console.log("Parent render. The number of points = " +  this.state.arr.length);
          var jsx =
                    <div>
                      <Displayer arr={this.state.arr}></Displayer>
                      <div>
                        <button id="buttonId"
                              onClick = {this.addOriginPoint}>Include Origin point
                        </button>
                      </div>
                    </div>;
           return jsx;
        }
}
```

The parent is syntactically correct BUT our re-render logic does not work the way we intended! Try adding the origin to the first array of points. The origin is not displayed; the parent does not re-render and so the child is also NOT re-rendered – excuse my grammar!

It looks like we ‘changed’ the parent state when we added the origin point and we did! The length of the array correctly went from THREE to FOUR. See console log.

Review: React re-renders a component when its state changes. React decides whether the state has indeed changed!

Issue: When we appended (push) a new item to our array React detected NO change because the state array attribute still referenced (‘pointed to’) the SAME/original array.

Example two

Change the parent from example one above.

From example one :

```
this.state.arr.push(new Point(0,0));
```

Remove the above line and replace with these two lines

```
let newArr = this.state.arr.concat([new Point(0,0)]);
this.setState({arr: newArr});
```

The java script array function concat returns a new array and so React detects a state change!

Example three

In example two we created a brand-new array to add the point (0,0). Quite a bit of work for such a small change! We try another approach.

Parent changes:

- Remove the array from the state. The array is now a plain class variable.
- Revert to example one where we added the origin via the push method.
- Create a new state with one attribute that toggles between 1 and -1
- Use the toggle logic to 'force' a re-render just after we push the origin

The parent constructor

```
constructor(props){
        super(props);

          this.addedOrigin = false;
          this.arr = [new Point(2,3), new Point (4,-9), new Point(4,0)];
          this.addOriginPoint = this.addOriginPoint.bind(this);
          this.state = {reRenderToggle:1}
}
```

The parent toggle logic

```
addOriginPoint(){

        if (this.addedOrigin === false){

           this.arr.push(new Point(0,0));

           this.setState({reRenderToggle : -this.state.reRenderToggle});

           console.log("Parent add origin. The number of points = " + this.arr.length);

           this.addedOrigin = true;
        } else{
           console.log("Parent add origin. Do nothing!");
        }
}
```

The parent render logic is basically the same except that the array is no longer part of the state.

We do not render anything from our state but we STILL re-render when the state changes!

```
render() {
        console.log("Parent render. The number of points = " + this.arr.length);
        var jsx =
        <div>
                <Displayer arr={this.arr}></Displayer>
                <div>
                  <button id="buttonId"
                        onClick = {this.addOriginPoint}>Include Origin point</button>
                </div>
        </div>;
        return jsx;
}
```

Example four

Here we use the state toggle idea from example three and apply it to a single Widget object versus an array.

```
export class Widget{
        // seven attributes
        #color = null;
        #model = null;
        #modelYear = null;
        #country = null;
        #price = null;
        #salePrice = null;
        #theme = null;

  constructor(){}

  // seven getter setter pairs

  getSalePrice(){   // getters for price reductions
        return this.salePrice;
  }

  setSalesPrice(x){
       this.salesPrice = x;
 }

  /// etc. setters and getters…
```

The Widget has NO knowledge of JSX. It is a simple entity. We add on some JSX magic by extending Widget in class Display Widget. The new class knows how to convert itself into an array of JSX list items- how convenient!

```
import React from 'react';
import {Widget} from './widget.js';export

class DisplayWidget extends Widget {

    toListItemArray (){

        let I = 0;
       let arr = [
            <li key = {I++}>Color:      {super.getColor()}</li>,
            <li key = {I++}>Model:      {super.getModel()}</li>,
            <li key = {I++}>Theme:     {super.getTheme()}</li>,
            <li key = {I++}>Year:       {super.getModelYear()}</li>,
            <li key = {I++}>Country:  {super.getCountry()}</li>,
            <li key = {I++}>Price:      {super.getPrice()}</li>,
            <li key = {I}>Sale Price:  {super.getSalePrice()}</li>,
        ];

        return arr;
    }
```

The parent creates and populates a Display Widget and supplies a price reducer!

```
import React from 'react';
import {DisplayWidget} from './displayWidget.js';
import {Displayer} from './displayer.js';
export class App extends React.Component{
        salesPercent = 0.95;// We reduce price in 5% intervals
        constructor(props){
            super(props);
            this.widget = new DisplayWidget();
            this.reducePrice = this.reducePrice.bind(this);
            this.populateWidget = this.populateWidget.bind(this);
            this.populateWidget();
            this.state = {reRenderToggle:1}
        }
        populateWidget(){
          this.widget.setColor('Blue');
          this.widget.setModel('Deluxe 2310');
          this.widget.setModelYear('2021');
          this.widget.setTheme('Rich and Famous');
          this.widget.setCountry('Singapore');
          let originalPrice = 12000;
          this.widget.setPrice(originalPrice);
          this.widget.setSalePrice(originalPrice);
        }

        reducePrice(){
          this.widget.setSalePrice(Math.round(this.salesPercent *
                this.widget.getSalePrice()));
          this.setState({reRenderToggle : -this.state.reRenderToggle});
        }

        render() {
                console.log(this.widget.getPrice());
                var jsx =
                <div>
                    <Displayer widget={this.widget}></Displayer>
                    <div>
                      <button onClick = {this.reducePrice}>Reduce price</button>
                    </div>
                </div>;
                return jsx;
        }
}
```

The Displayer

```
import React from 'react';

export const Displayer = (props) => {

    const items = props.widget.toListItemArray();
          console.log(items);
          return (
              <div>
                 <h2>Displaying a Widget</h2>
                 <ul> {items} </ul>
              </div> );
}
```

Render props

When a parent renders a child, the parent and child can share the same code.

Example one

In parent

this.widget = new Widget('Harry');

Parent renders <Child widget={this.widget}><Child>

Here the parent and child are sharing the same Widget instance.

Example two

The parent contains (uses) the function:

```
let widgetDisplay JSX = (widget) => {
   <p>{widget.toString()</p>
}
```

This function returns a React element (I say it returns valid **JSX**).

Parent renders

```
<Child widgetDisplayJSX ={this.widgetDisplayJSX }><Child>
```

Here the parent and child are sharing the same function.

The child can use the function to help render its JSX.

The child invokes the function via the line: this.props. widgetDisplayJSX (…).

Child render example

```
render(){
   return (<div>{ this.props. widgetDisplayJSX (…)}</div>
}
```

Example two above is an example of a '**render prop'** since the prop is a reference to a function that helps the child render. The function may provide the entire render logic for the child – or some the logic (a 'render-helper prop').

Example three

More child render examples. The child wraps the render function's return with <></> bracketing.

```
return <>{ this.props.render() }</>
```

A render prop does not have to be named '**render'**

```
return <>{ this.props.renderBuilder() }</>
```

A render prop function may require *arguments*. The child has the responsibility of providing the argument values, and the parent provides the render logic.

return <>{ this.props.renderBuilder(*this.age, this.name*) }</>

React ref: Grab a DOM element

Prior to my React days (like last week) I tried dabbling in Java Script.

Here are some Java Script classes that I used for drawing fun and games. Fun and games are always a good way to learn a new language. Each class represents one point. Note: a panel uses a horizontal & vertical axis with the origin at top-left corner.

```
export class PanelPoint {
  constructor(h,v) {
                this.h = h;
                this.v = v;
        }

        toString(){
                return ('(' +this.h+ ',' +this.v+ ')');
        }
}

export class XYPoint {
  constructor(x,y) {
                this.x = x;
                this.y = y;
        }

        toString(){
                return "("+this.x+","+this.y+")";

        }
}

// Angle is in radians
export class PolarPoint {

   constructor(len,angle) {
       this.len = len;
       this.angle = angle;
    }

    toString(){
                return ('(' +this.len+ ',' +this.angle+ ')');
        }
 }
```

I created some homemade drawing utilities as a ‘learning experience’. If you are not interested in the math just skip reading this class. Like any API we only need to call its public functions (interface) without knowing the gory details. It is important that the functions have ‘good’ descriptive and consistent names.

```
import {PanelPoint} from './panelPoint.js'
import {XYPoint} from './xyPoint'
import {PolarPoint} from './polarPoint'

export class SquareCanvasUtils {

   init(canvas) {
                this.canvas = canvas;
                this.canvasContext  = canvas.getContext('2d');
      this.halfsize = Math.floor(this.canvas.width/2);
      this.drawXYAxes();

      this.one_rad   = 180.0/Math.PI;
      this.rad_90  = Math.PI/2.0;
      this.rad_180 = Math.PI;
      this.rad_270 = 1.5*Math.PI;
      this.rad_360 = 0;
   }

   clearCanvas(){
      this.canvasContext.clearRect(0,0, this.canvas.width, this.canvas.height);
      this.canvasContext.beginPath();
      this.drawXYAxes();
   }

   degreeToRadians(D){
      return D*Math.PI/180.0
   }

   radiansToDegrees(radians){
      return radians*this.one_rad;
   }
```

```
polarPointToXY(P){
    let y = P.len*Math.sin(P.angle);
    let x = P.len*Math.cos(P.angle);
    let xyP = new XYPoint(x,y);

    return xyP;
}

xyToPolarPoint(P){
    let PP = new PolarPoint();
    if (P.x === 0 && P.y === 0){
        return new PolarPoint(0,0);
    }

    if (P.x === 0){
        var angle;
        if (P.y > 0){
            angle= this.rad_90;
        } else {
            angle = -this.rad_270;
        }
        return new PolarPoint(P.y, angle);
    }

    if (P.y === 0){
        if (P.x > 0){
            angle= 0.0;
        } else {
            angle = this.rad_180;
        }
        return new PolarPoint(P.x, angle);
    }

    // else do the trig
    PP.len = Math.sqrt(P.x * P.x + P.y * P.y);
    PP.angle = Math.atan(P.y/P.x);

    if (P.x < 0){
        PP.angle = Math.PI+PP.angle;
    }

    return PP;
}
```

```
drawXYPoint(xyP){
    let panelP = this.xyToPanelPoint(xyP);
    this.canvas.fillRect(panelP.h, panelP.v,1,1);
}

drawPolarPoint(polarP){
    let panelP = this.xyToPanelPoint(this.polarPointToXY(polarP));
    this.canvasContext.fillRect(panelP.h, panelP.v,1,1);
}

xyToPanelPoint(P){
    return new PanelPoint(this.halfsize+P.x, this.halfsize-P.y)
}

drawPanelLine(P1, P2){
    this.canvasContext.moveTo(P1.h, P1.v);
    this.canvasContext.lineTo(P2.h, P2.v);
    this.canvasContext.stroke();
}

drawXYAxes(){
    let X1 = new PanelPoint(this.halfsize, 0);
    let X2 = new PanelPoint(this.halfsize, 2*this.halfsize);
    let Y1 = new PanelPoint(0, this.halfsize);
    let Y2  = new PanelPoint(2*this.halfsize, this.halfsize);
    this.drawPanelLine(X1, X2);
    this.drawPanelLine(Y1, Y2);
}

XShift(points, delta){
    let shiftedPoints = points.map((point) =>{
        let x = point.x+delta;
        return new XYPoint(x, point.y)
     });

    this.connectXYPoints(shiftedPoints);
}
```

```
    YShift(points, delta){
        let shiftedPoints = points.map((point) =>{
            let y = point.y+delta;
            return new XYPoint(point.x, y)
         });

        this.connectXYPoints(shiftedPoints);
    }

    connectXYPoints(arr){
     for (let J = 0; J < arr.length-1; J++) {
        this.drawXYLine(arr[J],arr[J+1])
        }
    }

    drawXYLine(P1, P2){
        const _P1 = this.xyToPanelPoint(P1);
        const _P2 = this.xyToPanelPoint(P2);
        this.drawPanelLine(_P1,_P2)
    }
}
```

Back to React.

Look at this simple component. The JSX asks React to create HTML with a Canvas and React obeys. Run the application and observe our panel; not much to see!

```
import React from 'react';

export class Canvas extends React.Component{

    constructor(props){
        super(props);
        this.halfsize = 400;
    }

    componentDidMount(){
        console.log("We mounted!");
    }

   render(){

       const jsx =
         <canvas width={2*this.halfsize} height={2*this.halfsize}>
         </canvas>

       console.log("About to render a square canvas");
       return jsx;
    }
}
```

A lovely canvas is rendered but we cannot do anything with it! We now introduce the React Dom element reference. When asked, React provides us with a reference to any DOM element created via our JSX. The syntax is simple.

```
import React from 'react';

export class Canvas extends React.Component{

   constructor(props){
      super(props);
      this.halfsize = 400;
      this.domCanvas = null;
      this.canvasCapture = this.canvasCapture.bind(this);
   }

   canvasCapture(domCanvas){
      this.domCanvas = domCanvas;
   }

  render(){

    const jsx =
      <canvas width={2*this.halfsize} height={2*this.halfsize} ref={this.canvasCapture}>
      </canvas>

      console.log("About to render a square canvas");
      return jsx;
   }
}
```

The keyword **ref** instructs React to grab a reference to the corresponding DOM element (here the canvas) and send it as an argument to the user provided function – here the **canvas capture**. React converts the JSX to HTML and provide us with a reference to the canvas!

Here is the plan.

Our canvas component is responsible for rendering the canvas and grabbing a reference to the Dom canvas element. The component takes the canvas reference and passes it to our Java Script class **Square Canvas Utils**. The canvas component is NOT a talented artist! So, it sends the **utils** as a 'paint tools' prop to an Artist child.

The Artist follows.

```
import React from 'react';
import {PolarPoint} from './polarPoint.js';
import {XYPoint} from './xyPoint.js'

export class Artist extends React.Component{

  constructor(props){
     super(props);
     this.paintTools = this.props.paintTools;

     this.clearCanvas = this.clearCanvas.bind(this);
     this.drawCircle = this.drawCircle.bind(this);
     this.drawSquare = this.drawSquare.bind(this);
     this.initBottle = this.initBottle.bind(this)
     this.rotateBottleRight = this.rotateBottleRight.bind(this);
     this.rotateBottleLeft = this.rotateBottleLeft.bind(this);
     this.rotateBottle = this.rotateBottle.bind(this);

     this.activePolarBottle = [];
     this.bottle = [];
     this.bottleRotationDelta = 5.0*Math.PI/180.0

     this.initBottle();
  }

  drawSquare(evt){
     const side = Math.abs(evt.target.value);
     let points = [];
     var p1;
     p1 = new XYPoint(side,side,50);
     points.push(p1);
     points.push(new XYPoint(-side,side));
     points.push(new XYPoint(-side,-side));
     points.push(new XYPoint(side,-side));
     points.push(p1);
     this.paintTools.connectXYPoints(points);
   }

  drawCircle(evt){
     const radius = Math.abs(evt.target.value);
     for (var D = 0; D <= 360; D++){
        let polarP = new PolarPoint(radius, this.paintTools.degreeToRadians(D));
        this.paintTools.drawPolarPoint(polarP);
     }
   }
```

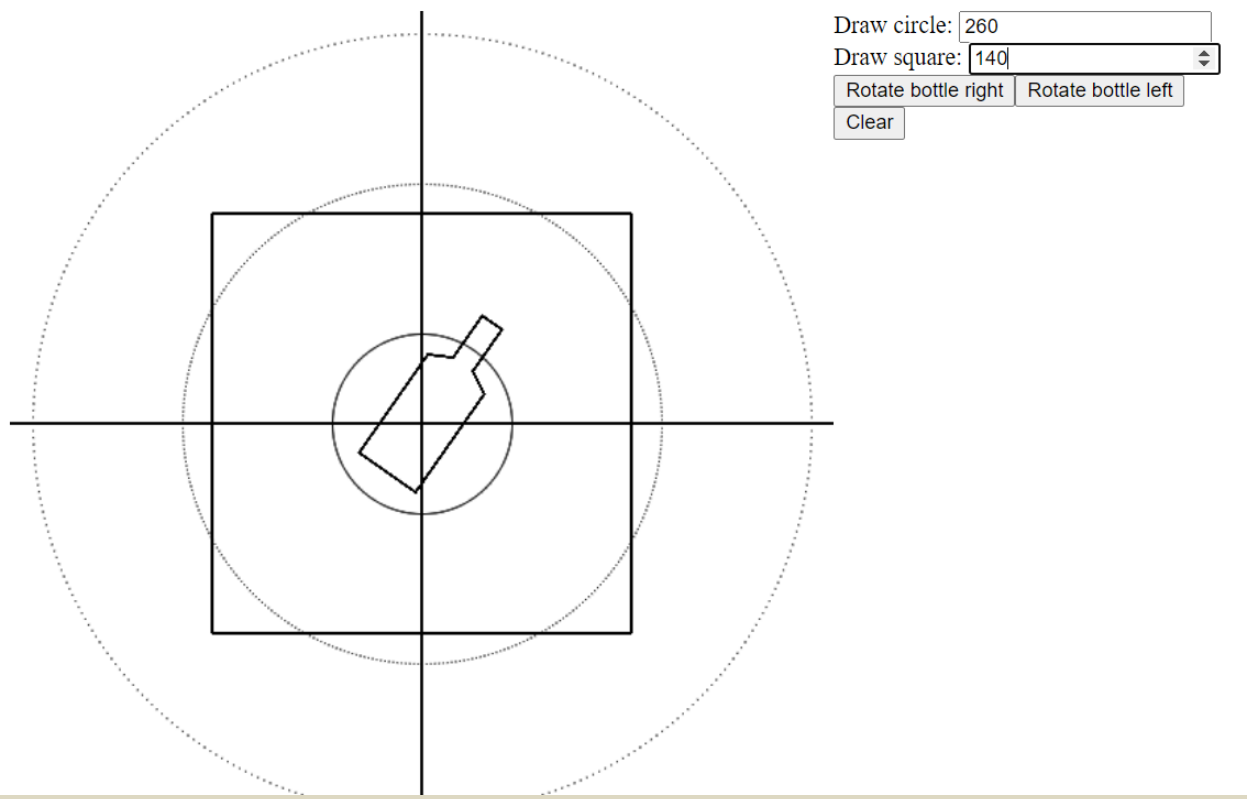
```
rotateBottleLeft(){
  this.rotateBottle(1);
}

rotateBottleRight(){
  this.rotateBottle(-1);
}

rotateBottle(DIRECTION){

  this.paintTools.clearCanvas();
  var delta;

  if (this.activePolarBottle.length === 0){

    this.activePolarBottle = this.bottle.map((xyP) => {
      return this.paintTools.xyToPolarPoint(xyP)});

    delta = 0;
  } else {
    delta = DIRECTION*this.bottleRotationDelta;
  }

  this.activePolarBottle.forEach((PP) => {PP.angle = PP.angle + delta});

  let bottleToDisplay = this.activePolarBottle.map((pp) => {return
      this.paintTools.polarPointToXY(pp)});

  this.paintTools.connectXYPoints(bottleToDisplay);
}

initBottle(){
  var bottle = this.bottle;
  bottle.push(new XYPoint(23,-40));
  bottle.push(new XYPoint(23,40));
  bottle.push(new XYPoint(8,48));
  bottle.push(new XYPoint(8,82));
  bottle.push(new XYPoint(-8,82));
  bottle.push(new XYPoint(-8,48));
  bottle.push(new XYPoint(-23,40));
  bottle.push(new XYPoint(-23,-40));
  bottle.push(new XYPoint(23,-40));
}
```

```
  clearCanvas(){
    this.paintTools.clearCanvas();
    this.activePolarBottle.length = 0;
  }

  render(){
    const jsx =
    <div>
      Draw circle: <input type="number" onChange={this.drawCircle}/>
      <br></br>
      Draw square: <input type="number" onChange={this.drawSquare}/>
      <br></br>
      <button
          onClick={this.rotateBottleRight}>Rotate bottle right
       </button>
       <button
          onClick={this.rotateBottleLeft}>Rotate bottle left
       </button>
       <br></br>
       <button
          onClick={this.clearCanvas}>Clear
       </button>
    </div>

    return jsx;
  }
}
```

The canvas component.

```
import React from 'react';
import {SquareCanvasUtils} from './squareCanvasUtils.js'
import {Artist} from './artist.js'

export class Canvas extends React.Component{

   constructor(props){
      super(props);
      this.halfsize = 275;
      this.domCanvas = null;
      this.canvasCapture = this.canvasCapture.bind(this);
      this.canvasUtils = new SquareCanvasUtils();
   }

   canvasCapture(domCanvas){
      this.domCanvas = domCanvas;
      this.canvasUtils.init(domCanvas);
   }

   render(){

      const jsx =
      <div style= {{display:"flex", flexDirection:"row"}}>
         <div>
         <canvas width={2*this.halfsize} height={2*this.halfsize}
                  ref={this.canvasCapture}>
         </canvas>
         </div>

         <div>
         <Artist paintTools={this.canvasUtils}></Artist>
         </div>
      </div>
      return jsx;
   }
}
```

Run the application and play. Below I rotated my bottle just a little; I did not want to spill any wine! I added a few circles and a square. Notice the larger circle issues as we do not draw enough points. See the **draw circle** artist function! Try to implement logic to increase the number of points as a function of the radius.



If there are bug issues. Try removing the artist from the Canvas and test the drawing logic in the Canvas component. A good place is after the mount.

```
componentDidMount(){

      // Place temp test code here
      // e.g., Run for XY  (4,400) (-4,400) (4,-400) (-4, -400)  (0, 10) (0, -10) (10,0) (-10,0)

      let XY = new XYPoint(5, -100);
      console.log("Before " + XY.toString());
      let PP =  this.canvasUtils.xyToPolarPoint(XY);
      console.log("Polar " + PP.toString());
      let XY2 = this.canvasUtils.polarPointToXY(PP);
      console.log("After " + XY2.toString());

     // etc…
  }
```

The Use State Hook

React 16.8.0 introduced the concept of Hooks. A Hook is an API to help make your (the React programmer) life easier.

Hooks are used exclusively in ***function components***. Hooks do **not** 'work' in *class components*.

We have seen that the state in a *class component* is the key to keeping our HTML pages up to date when a change is made to data. React detects a state change and invokes the components render method (term: re-render or update) to create a new bunch of HTML that hopefully reflects the changes in the data.

We now look at the *component state* related Hook. What!? A function component is just a java script function that returns JSX. How can such a component have a persistent data (state) across multiple invocations? The answer is the React useState API.

The function component contains the java script logic needed to create and return valid JSX. The related state and state changes are persisted under the hood by React. When the function component is rendered N times the function is invoked (called) N times and the same state is available for each invocation., persisted between invocations

Example one.

```
import React from 'react';
import {Child} from './child.js';

export class App extends React.Component{
   constructor(props){
      super(props);
      this.reRenderParent = this.reRenderParent.bind(this);
      this.state = {count:1}
   }

   reRenderParent(){
      this.setState({count : this.state.count+1});
   }

   render(){
     return (
      <div>
         <p>Parent render counter: {this.state.count}</p>
         <Child counter={this.state.count}></Child>
         <div>
         <button id="buttonId"
         onClick = {this.reRenderParent}>Parent re-render</button>
         </div>
      </div> );
   }
}
```

We add the render count to the parent and child for demonstration/debugging.
We want to know the number of times rendering occurs

The child follows.

```
import React from 'react';

export const Child = (props) => {

  console.log("Child function called....");

  // State initialization

   const [detailsFlag, setDetailsFlag] = React.useState(true);
   const [mood, setMood] = React.useState('Happy');
   const [details, setDetails]   = React.useState({name:'Jose', age:34, hobby: 'chess'});

   const toggleDetails = () => {
      setDetailsFlag(() => (!detailsFlag))
   }

   const changeMood = () => {
         if (mood ==='Happy'){
          setMood(() => ('Sad'));
      }else{
          setMood(() => ('Happy'));
      }
   }

   const performBirthday = () => {
      setDetails((prev) => ({name: prev.name, age: prev.age+1,  hobby:prev.hobby}));
    }

   const detailsJSX =
      <div>
       <p>Name: {details.name} Age: {details.age} Hobby: {details.hobby} Mood: {mood}
       </p>
      </div>

   const  noDetailsJSX =
     <div>
        <p>Name: {details.name}
      </p>
   </div>
```

```
return (
      <div>
          <p>Counter from parent: {props.counter}</p>

           {detailsFlag ? detailsJSX : noDetailsJSX }
           <br></br><br></br>

          <button onClick={toggleDetails}>Toggle Details</button>
          <br></br>

          <button onClick={performBirthday}>Birthday</button>
          <br></br>

         <button onClick={changeMood}>Mood</button>

      </div>
   )
}
```

Notes:

We invoke useState multiple times.

These are valid variations of our toggle function

```
const toggleDetails = () => {
   setDetailsFlag(!detailsFlag)
}
```

```
const toggleDetails = () => {
   setDetailsFlag(() => { return !detailsFlag})
}
```

Play with the various buttons and notice the parent and child re-renders and how child data is persisted for N function invocations (re-renders)

Example two

Child

```
import React from 'react';
export const Child = (props) => {

   const [details, setDetails]   = React.useState({age:34, hobby: 'chess'});

   const performBirthday = () => {
      setDetails((prev) => ({age: prev.age+1,  hobby:prev.hobby}));
    }

   console.log("Child "  + props.name + " my age is : " + details.age);
   return (
          <div>
          <p>My parent named me: {props.name}</p>
          <p>My age is: {details.age}</p>
          <p>My hobby is: {details.hobby}</p>
          <button onClick={performBirthday}>Birthday for {props.name}</button>
          <br></br>
          </div>
   )
}
```

Parent

```
import React from 'react';
import {Child} from './child.js';

export class App extends React.Component{
   constructor(props){
      super(props);
      this.reRenderParent = this.reRenderParent.bind(this);
      this.state = {count:1}
   }

   reRenderParent(){
      this.setState({count : this.state.count+1});
   }

   render(){
     return (
      <div>
         <p>Parent render counter: {this.state.count}</p>

         <div>
         <Child name="Wendy"></Child>
         </div>
         <div>
         <Child name="Willy"></Child>
         </div>
         <div>

         <br></br>
                  <button id="buttonId"
                  onClick = {this.reRenderParent}>Parent re-render</button>
                  </div>
      </div> );
   }
}
```

Run the application and click away. Notice that Wendy and Willie have their own (separate) persistent states.

Dressed up for New Year's Eve

We create a java script class for an Ensemble a.k.a. **outfit** (what to wear) and two components App (parent) and Ensemble Displayer. The parent creates an outfit ('hard coded' here for simplicity) and asks the displayer to display **and critique**. The critique requirement will cause a bit of an issue which we will fix!

Related java script classes

```
export class EnsembleItem {
    #article;  #color;

    constructor(article, color) {
        this.article = article;
        this.color = color;
    }
    getArticle(){
        return this.article;
    }
    getColor(){
        return this.color;
    }
}
```

```
import {EnsembleItem} from './ensembleItem.js'

export class Ensemble {
    #description;   #items;

   constructor(description) {
        this.description = description;
        this.items=[];
    }

    addItem(article, color){
        const item = new EnsembleItem(article, color);
        this.items.push(item);
    }

    getDescription(){
        return this.description;
    }

    getItems(){
        return this.items;
    }
}
```

The Displayer

```
import React from 'react';

export class EnsembleDisplayer extends React.Component{

    constructor(props){
        super(props);
        this.buildItemList = this.buildItemList.bind(this);
    }

    buildItemList(){
      const items = this.props.ensemble.getItems();
      let I=0;
      const list = items.map(item =>
        {
            let itemStr =  "Article: " + item.getArticle() + " Color: " + item.getColor();
            let line = <li key={I++}>{itemStr}</li>;
            return line;
        });

       return list;
    }

    render(){
        const itemList = this.buildItemList();

        const jsx = <div>
            <h2>The ensemble {this.props.ensemble.getDescription()}</h2>
            <ul>
                {itemList}
            </ul>
            <h2>The ensemble critique</h2>
            <p>{this.props.ensembleCritique}</p>
        </div>

        return jsx;
    }
}
```

Run this and notice that we have an issue! The displayer is expecting a Critique in its props BUT the parent component does not 'know how' to critique an ensemble . The displayer wants to render this:

```
<p>Critique : {this.props. ensembleCritique}</p>
```

Goal: fix this issue ***without*** changing the outfit displayer. I repeat ***without!!!***

Approach One: Writing and using a HOC

Displayers should be as dumb and simple as possible; the function is simply to display something.

Our HOV creates and returns a new component. The new component takes the existing Displayer definition as an input and creates a new component that takes (via props) the App provided ensemble , critiques the ensemble , and finally sends both the ensemble and the critique to its Displayer!

Hum that may sound a bit complicated. It is NOT and an example will illustrate just how simple it is!

Term: A **Higher order component** (HOC) is a java script function that takes a component definition as an input and returns a (another) component definition.

C2 = function(C1) where C1 and C2 are components.

HOC Input: a component definition (**C1**)

HOC function builds and returns an anonymous component (**C2**) definition.

Note: both C1 and C2 rare variables whose values are ***references*** to two React **component definitions**. Such a definition is similar in nature to a JAVA or C++ Class as it is a template for creating actual component instances.

Review: Our top level (parent) App component definition is rendered in the index.js file via the call to ReactDOM.render.

In our example ***C2*** expects a prop containing an ensemble; **C2** extracts the ensemble's colors and performs a ***critique***. **C2** may (will!) receive props that contain ***additional information*** that **C2** does not need to perform its function!

C2 render**: C2** renders the **C1** component sending **C1** TWO props:

1. **C2's** props contain the ensemble and possibly other ***additional information***
2. The above ***critique***

Since I am a bit color blind, we keep the critiquing logic rather primitive. The **HOC** uses our java script helper function ColorCriticFunc where the actual critiquing logic is encapsulated. The HOC renders the **C1** component.

First, the stand-alone critique logic – a bit silly!

```
export function ColorCriticFunc(ensemble){
    const BLACK = 'BLACK';
    const WHITE = 'WHITE';

    const colors = ensemble.getItems().map(item => item.getColor());
    let colorSet = new Set();

    for (let i = 0; i < colors.length; i++) {
        colorSet.add(colors[i].toUpperCase());
    }

    const setSize = colorSet.size;

    let allBlackWhite = setSize === 2 && colorSet.has(WHITE) && colorSet.has(BLACK);

    if (allBlackWhite){
        return "A very Chic Black and White Ensemble"
    }

    let allBlack = setSize === 1 && colorSet.has(BLACK);

   if (allBlack){
        return "A very Chic All Black Ensemble"
    }

    let allDifferent = setSize === colors.length;

    if (allDifferent){
        return "A very Colorful Ensemble"
    }

    // not all black and all same color
    if (setSize === 1){
        return "A very BLAND Ensemble"
    }

    return "No Critique is available for this Ensemble"
}
```

The HOC function.

```
import React from 'react';
import {ColorCriticFunc} from './colorCriticFunc.js';

export const ColorCriticHOCFunc = (InnerComponent) => {
   // C2 = ColorCriticHOCFunc(C1)
   //   create and return C2
   //   where InnerComponent is C1

   return class extends React.Component {

      critique(){
         const ensemble= this.props.ensemble;
         const theCritique = ColorCriticFunc(ensemble);
         return theCritique;
      }

      render(){
         let critique = this.critique();
         return (
         <InnerComponent
            {...this.props} ensembleCritique={critique}>
         </InnerComponent>);
      }

   } // end of anonymous class definition

} // end of outermost function
```

Our helper ColorCriticFunc allows us to keep the ugly critiquing logic out of the HOC so that we can zero in on the HOC related logic of creating and returning the new component (**C2**) definition

The EnsembleDisplayer does not change. It stays dumb per our requirement!

The App

```
import React from 'react';
import {Ensemble} from './ensemble.js' ;
import {EnsembleDisplayer} from './ensembleDisplayer.js' ;
import {ColorCriticHOCFunc} from './colorCriticHOCFunc.js' ;

export const  App = () => {

    // C1 is our EnsembleDisplayer
    // ColorCriticHOCFunc is our HOC function
    // C2 (the HOC function output) is our MyEnsembleDiplayer

    //invoke (call) the HOC
    const MyEnsembleDiplayer = ColorCriticHOCFunc(EnsembleDisplayer);

    const anEnsemble = new Ensemble ("New Years Eve in Times Square")
    anEnsemble.addItem('coat', 'Red');
    anEnsemble.addItem('scarf', 'White');
    anEnsemble.addItem('gloves', 'Blue');

    return (
    <div>
        <MyEnsembleDiplayer ensemble={anEnsemble}/>
    </div>);
}
```

Notes:

- The parent invokes the Color Critic HOC Func with an argument of type Ensemble Displayer
- The HOC function internally uses the generic name Inner Component for the above input argument
- The HOC returns a definition of an anonymous component that ‘wraps’ its Inner Component
- The parent assigns the anonymous component to the variable name: My Ensemble Displayer.
- The parent renders the My Ensemble Displayer and sends it an Ensemble
- The My Ensemble Displayer component reads the parent provided props (the Ensemble) and critiques it
- The anonymous component then renders the child Inner Component which is an Ensemble Displayer with **all** the props stuff expected by the child.
- Notice how the anonymous component uses the java script *spread operator* (a.k.a. dot dot dot)

Approach Two: Critiquing via a java script function

The above HOC in ***Approach One*** has very limited render 'smarts' except to know what props to send its child Inner Component. The heavy work is the color critique functionality which we already encapsulated in the simple java script function Color Critic Func.

This change is simple. We remove (toss it!) the HOC function Color Critic HOC Func and change the App component.

The App.

```
import React from 'react';
import {Ensemble} from './ensemble.js' ;
import {EnsembleDisplayer} from './ensembleDisplayer.js' ;
import {ColorCriticFunc} from './colorCriticFunc.js' ;

export const  App = () => {

   const anEnsemble = new Ensemble ("New Years Eve in Times Square")
   anEnsemble.addItem('coat', 'Red');
   anEnsemble.addItem('scarf', 'White');
   anEnsemble.addItem('gloves', 'Blue');

   const ensembleCritique = ColorCriticFunc(anEnsemble);

   return (
   <div>
      <EnsembleDisplayer
         ensemble={anEnsemble} ensembleCritique={ensembleCritique}>
      </EnsembleDisplayer>
   </div>);
}
```

Notes:

- We change the parent to call the Color Critic Func function directly which returns a string – the critique
- We change the parent to directly render the Ensemble Displayer with the complete set of required props!

I am not one to give advice but if I was I would recommend writing and using a HOC (Approach one) **only** when it is simpler and cleaner then Approach two!

A brief discussion about asynchronous functions

This chapter may be a bit boring to seasoned programmers!

Our java script functions have been synchronous. A function call is made, and the caller waits for the function to return. In some cases, the wait may be a long time, and this is an issue for our web end-user

Suppose a master house painter Judy is working with an apprentice assistant Felix. Judy has a list (FIFO, First In First Out Queue) of Tasks to complete. She can only do one Task at a time and does NOT want to be interrupted in the middle of performing a Task! If the list is empty she waits until the list is not empty!

Judy pops the following ***Task$_1$*** from her queue: assign work for Felix. Felix is given the work to spackle the den room a ***Task$_2$***. Judy's ***Task$_1$*** is to:

Assign the work to Felix and gives Felix appropriate instructions.

Felix, in turn:

- Felix promises to report back to Judy when his work is completed
- Felix promises to report any issues/problems encountered – if any!

Judy is done with the ***Task$_1$*** and pops the next one to work on. She does **not** wait for Felix to finish – why would/should she?!

Felix:

- Felix keeps the promise, e.g., Felix does the assigned work and does **not** go directly to the local pub after the spackling - instead he reports.
- Felix reports back by placing a new ***Task$_3$*** in Judy's list.

When Judy gets around to popping Felix's ***Task$_3$*** she has hold of the completed report and takes appropriate action. Then, Judy continues popping Tasks and working on them.

Remote server fetch via XML Http Request

Install Postman

On your browser go to the following URL to get a nice random token (a UUID); we will call it *<YOUR TOKEN>*

https://www.uuidgenerator.net/

If you do not have Chrome's Postman install it. Open Postman, pick GET and type in the following URL with the above token

https://gorest.co.in/public/v1/users?access-token=*<YOUR TOKEN>*

JSON formatted data is returned. Look at the JSON; notice the array (square brackets) labeled '**data'**.

The code follows. The following class will house one line in the array 'data' that is returned from our URL. Compare it with the Postman results.

```
export class RemoteDataItem {

constructor(j){
    this.id  = j.id;
    this.name = j.name;
    this.email  = j.email;
    this.gender = j.gender;
    this.status  = j.status
}

  toString() {
      return "" + this.name +  " " + this.email;
   }
}
```

The React App component constructor.

```
constructor(props){
        super(props);

        this.state = {renderToggle: 1};
        this.dataArr = [];
        this.request =  new XMLHttpRequest();

        // the callback
        this.processResponse = this.processResponse.bind(this);
        this.forceRender = this.forceRender.bind(this);
        this.buildDataList = this.buildDataList.bind(this);
        this.sendGETRequest = this.sendGETRequest.bind(this);
    }
```

- The above **data array** is our container for N Remote Data Items objects derived via JSON.
- The **request** is sent to the remote server (to the URL).
- The **process response** is our callback function.

The following function builds a list item <li> for each element in our data array.

```
buildDataList(){

      let I=0;
      const list = this.dataArr.map(remoteDataItem =>
      {
         let itemStr = remoteDataItem.toString();
         let line = <li key={I++}>{itemStr}</li>;
         return line;
      });

      return list;
  }
```

The following function forces a re-render. We need this when the data arrives and the above <li> is built. It simply performs a state change.

```
forceRender(){
      this.setState({reRenderToggle : -this.state.reRenderToggle});
   }
```

The following sends the GET request. The process response function is our callback. This **send** is asynchronous; it returns before the remote host responds back.

```
sendGETRequest(){
        var URL =
        "https://gorest.co.in/public/v1/users?access-token=<YOUR TOKEN>";

        this.request.open("GET", URL, true);
        this.request.addEventListener("readystatechange", this.processResponse, false);
        this.request.send();  // spawns a new thread
    }
```

The following is our call back logic. We **populate** our data array and **re-render**.

```
processResponse(){
        console.log("Callback: state: "+
                    this.request.readyState+" status: "+this.request.status);

        if ((this.request.readyState === 4 ) && (this.request.status === 200)){
            var str = this.request.responseText;
            var obj = JSON.parse(str);
            var rows = obj.data;   // we have knowlegde re the repsonse JSON format

            for (var i = 0; i < rows.length; i++) {
                let remoteData = new RemoteDataItem(rows[i]);
                this.dataArr.push(remoteData);
            }

            this.forceRender();
        }
    }
```

We use our knowledge of the JSON data formatting; look at the Postman output. The data subset we are interested in is the JSON array labeled 'data'. The JSON is represented by a java script ***string*** in the request's response text.

This JSON ***string*** data is converted into an ***array*** of java script objects each resembling the JSON data labeling (label, value) pairs. This conversion is done by the JSON.parse() function.

Finally, each item of the above ***array*** is sent to our Remote Data Item constructor, and the constructed object is saved in our **data array** – an array of **Remote Data Item.**

Once the **data array** is populated we need to render it. We force a render.

We are almost done! In case you have not noticed we have not made the GET call yet. The question is where in the code should we put it? The logic in render method is a clue!

```
render(){
      var retJSX;

      if (this.dataArr.length ===0){
         retJSX = <div>Loading Data please be patient </div>
      } else {
         retJSX = <div>
                        <ul>
                          {this.buildDataList()}
                        </ul>
                    </div>
      }

      return retJSX;
   }
```

When the application first comes up we want to immediately show something to our end user. We have no data yet and the fetch to the remote URL may take a long time. We must assume it will!

Our first render therefore will display: **Loading Data please be patient.**

When the data **arrives**, our callback populates our data array and forces a re-render; this time with a bunch of data! But the data will never **arrive** unless we send the GET!

The solution: make the GET call is just after our initial render (the mount : see the chapter: Component Class Life cycle).

We code:

```
componentDidMount(){
      this.sendGETRequest();
  }
```

Notes:

See the console logging at the top of our callback function. Run the application. Examine the log and you will see that the callback was invoked more than once! I see three log entries.

Callback: state: 2 status: 200
Callback: state: 3 status: 200
Callback: state: 4 status: 200

We process the data (it has fully arrived) when the **state** is 4 and the **status** is 200

Google 'html status'. The status 200 indicates that everything is OK so far! The status may indicate some issue – something BAD like 401 Unauthorized!

The state changes. In chronological order:

State	Indicates
2	send() has been called, and headers and status are available
3	Downloading; response text holds **partial** data.
4	The operation is complete.

For very long GET calls you can imagine performing re-renders each time a new <state, status> pair is provided. You could render information to the end user regarding the status of the Data Load!

Our example has no error checking; it is clearly not intended as an example of a ‘production ready’ application!

Remote server fetch with async/await

We will repeat the previous chapter using the java script asych/await

Merriam-Webster provides these definitions

Wait

1a: to remain stationary in readiness or expectation *wait* for a train
b: to pause for another to catch up —usually used with *up*

2a: to look forward expectantly just *waiting* to see his rival lose
b: to hold back expectantly *waiting* for a chance to strike

Await

1a: to wait for We are *awaiting* his arrival. *await* a decision
b: to remain in abeyance until a treaty *awaiting* ratification

Abeyance

1: a state of temporary inactivity : suspension

We take the example from the previous chapter and replace the HTTP approach with the java script async/await logic and the java script fetch method. We also add related error checking.

The **fetch** function is part of the java script **Fetch API**.

Constructor: we include a new logic to handle fetch and related exceptions

```
constructor(props){
      super(props);

      this.state = {renderToggle: 1};
      this.dataArr = [];
      this.fetchIssue = null;
      this.renderCount = 0;
      this.processResponse       =  this.processResponse.bind(this);
      this.forceRender           =  this.forceRender.bind(this);
      this.buildDataList         =  this.buildDataList.bind(this);
      this.sendGETRequest        =  this.sendGETRequest.bind(this);
      this.handleFetchException =  this.handleFetchException.bind(this);
   }
```

The following two functions do not change

```
buildDataList(){

      let I=0;
      const list = this.dataArr.map(remoteDataItem =>
      {
         let itemStr = remoteDataItem.toString();
         let line = <li key={I++}>{itemStr}</li>;
         return line;
      });

      return list;
  }

forceRender(){
      this.setState({reRenderToggle : -this.state.reRenderToggle});
   }
```

The process response function changes.

```
processResponse(responseText){

        var responseObj = JSON.parse(responseText);
        var rows = responseObj.data;   // we have knowlegde re the repsonse JSON

        for (var i = 0; i < rows.length; i++) {
            let remoteData = new RemoteDataItem(rows[i]);
            this.dataArr.push(remoteData);
        }
        this.forceRender();
   }
```

Send GET request

```
async sendGETRequest() {
      var URL =
      "https://gorest.co.in/public/v1/users?access-token=<YOUR TOKEN>";

      let response = await fetch(URL);
      if (!response.ok){
          let issue = 'ERROR: HTTP issue. Status: ' + response.status;
          throw new Error(issue);
      }
      return await response.text();
   }
```

Our send request may throw an Error; we catch any Error and **handle it**.

```
componentDidMount(){

    this.sendGETRequest().then (
        (responseText) => {
            this.processResponse(responseText);
        }
    ).catch((e) => {this.handleFetchException(e);});
 }
```

We notify end user of any ‘issue’ encountered via the function. Here is an instance of a java script Error

```
 handleFetchException(e){
      this.fetchIssue = e;
      this.forceRender();
   }

   render(){

      var retJSX;

      if (this.fetchIssue){

         let errorMsg =  'Loading Data Error occurred Call IT now! Provide the msg: '
               + this.fetchIssue;

          console.log(errorMsg);

          retJSX = <div>{errorMsg}</div>

      } else

         if (this.dataArr.length ===0){
              retJSX = <div>Loading Data please be patient</div>
         } else {
             retJSX = <div><ul>
               {this.buildDataList()}
               </ul></div>
      }

      return retJSX;
   }
}
```

Run the application. Use the URL:

"https://gorest.co.in/public/**v1**/users?access-token=*YOUR TOKEN>*

Now modify the above URL in the code --> change **v1/** to **vX1/**
This change should force an error case for you to test.

The following version of the code also works. It more strongly illustrates that an **async labeled function** returns a Promise object – too bad java script was not more strongly 'typed'!

```
componentDidMount(){

    let promise = this.sendGETRequest();

    promise.then (
        (responseText) => {
            this.processResponse(responseText);
        }
    ).catch((e) => {this.handleFetchException(e);});
}
```

Make sure the application is working before proceeding!

We now add a bunch of time stamped logging to see the timeline.

To the constructor add:

```
        this.logArr = [];
        this.logger = this.logPromise.bind(this);
```

Add this new function

```
logPromise(msg, promise){
        let _now =new Date().getTime();

        if (promise instanceof Promise){
            this.logArr.push({time: _now, msg: msg, note: 'This is a Promise'});
        } else {
            this.logArr.push({time: _now, msg: msg, note: null});
        }
    }
```

Add logging to the process response function

```
processResponse(responseText){
    this.logPromise('Function processResponse : Top');

    var responseObj = JSON.parse(responseText);
    var rows = responseObj.data;   t

    for (var i = 0; i < rows.length; i++) {
            let remoteData = new RemoteDataItem(rows[i]);
            this.dataArr.push(remoteData);
    }

    this.logPromise('Function processResponse about to force render and return');
    this.forceRender();
 }
```

Adding logging to the componentDidMount function

```
componentDidMount(){

    this.logPromise('Function ComponentDidMount: TOP');
    this.logPromise('Function ComponentDidMount: Call sendGETRequest');

    this.sendGETRequest().then (
    (responseText) => {
       this.logPromise('Function componentDidMount" entered THEN. Data arrived');
       this.processResponse(responseText);
    }
    ).catch((e) => {this.handleFetchException(e);});

    this.logPromise('Function ComponentDidMount: returning');
}
```

We dump the log data to the console after the second render, i.e., after the first and only update.

```
componentDidUpdate(){
       console.log('Function async componentDidUpdate: Top about to dump log');
       let prev = this.logArr[0].time;
        this.logArr.forEach(
            item=>{
                let elapsed = item.time - prev;
                if(item.note === null){
                   console.log('Elapsed ms: ' + elapsed  +   " " + item.msg);
                }  else {
                   console.log('Elapsedms: ' + elapsed + " "  + item.msg + " " + item.note);
                }
                prev = item.time;
            });
   }
```

Adding logging to the send GET request function

```
async sendGETRequest() {

   this.logPromise('Function async sendGETRequest: Top async get');

   var URL =
   "https://gorest.co.in/public/v1/users?access-token=<YOUR_TOKEN>;

   this.logPromise(
         'Function async sendGETRequest:  about to call await fetch function');

   let responsePromise = await fetch(URL);

   this.logPromise(
         'Function async sendGETRequest:  returned from await wrappedFetch' ,
         responsePromise);

   if (!responsePromise.ok){
          let issue = 'ERROR: HTTP issue. Status: ' + responsePromise.status;
          throw new Error(issue);
   }

   //wait until text() finished
   let retText = await responsePromise.text();

   this.logPromise('Function async sendGETRequest: :  returning txt.', retText );
   return retText;
}
```

Add logging to render.

```
render(){
       this.logPromise('Function render:  TOP');
       …
       this.logPromise('Function render: about to return');
       return retJSX;

}
```

Run the application. In the console log I see the following. Does the order of events make sense?

Function async componentDidUpdate: Top about to dump log

Elapsed ms: 0 Function render: TOP
Elapsed ms: 0 Function render: about to return
Elapsed ms: 5 Function ComponentDidMount: TOP
Elapsed ms: 0 Function ComponentDidMount: Call sendGETRequest
Elapsed ms: 1 Function async sendGETRequest: Top async get
Elapsed ms: 0 Function async sendGETRequest: about to call await fetch function
Elapsed ms: 1 Function ComponentDidMount: returning
Elapsed ms: 705 Function async sendGETRequest: returned from await fetch
Elapsed ms: 1 Function async sendGETRequest: : returning txt.
Elapsed ms: 0 Function componentDidMount" entered THEN. Data arrived
Elapsed ms: 0 Function processResponse : Top
Elapsed ms: 0 Function processResponse about to force render then return
Elapsed ms: 1 Function render: TOP
Elapsed ms: 1 Function render: about to return

Another SPA

Using test data

Earlier we looked at a skeleton SPA example. In this chapter we create an SPA with some muscle.

There are two entities placed in the ***./src/entities*** folder

```
export class ToDoItem {
    userId = null;
    item = null;
    due = null;
    status = null;

    constructor(j){
        if (j){
            this.userId = j.user_id;
            this.item   = j.title;
            this.due    = j.due_on;
            this.status = j.status;
        }
    }
}
```

```
export class User {
    id = null;
    name = null;
    email = null;
    status = null;

    constructor(j){
        if (j){
        this.id = j.id;
        this.name = j.name;
        this.email = j.email;
        this.status = j.status;
        }
    }
}
```

We create three SPA related ‘pages’

1. home (our default page)
2. users page
3. to dos page

In the home page we hard code the data. We **prevent** updates (re-renders)

```
import React from 'react';

export class Home extends React.Component {

    constructor(props){
        super(props);
        this.buildList = this.buildList.bind(this);
    }

    buildList(){
        let I = 0;
        let list = [];
        list.push(<li key={I++}>CEO: R Popcorn Callaway</li>);
        list.push(<li key={I++}>VP: J Callaway</li>);
        list.push(<li key={I++}>lead Tester: W Max</li>);
        list.push(<li key={I++}>Lead Designer: S Sam</li>);
        list.push(<li key={I++}>Junior Programmer: L Adams</li>);
        list.push(<li key={I++}>Office Manager: JJ Lopez</li>);
        return list;
    }

    shouldComponentUpdate(){
        return false;
    }

    render(){
        return (
            <div>
                <h2>Welcome to Cut and Paste Ink</h2>
                <p>Employee list</p>
                <ul>
                {this.buildList()}
                </ul>
                </div>
        );
    }
}
```

```
import React from 'react'
import "./CSS/tableStyles.css"
export class UsersPage extends React.Component {
    constructor(props){
        super(props);
        this.buildHdr = this.buildHdr.bind(this);
        this.buildRows = this.buildRows.bind(this);
    }

    buildRows(){
         let I = 0;
         const list = this.props.users.map(u => {
          let oneRow = <tr className='rowStyle' key = {I++}>
             <td>{u.id}</td> <td>{u.name}</td> <td>{u.email}</td> <td>{u.status}</td></tr>
            return oneRow;
    });

    return list;
    }

    buildHdr(){
           let JSX =
            <tr className='rowStyle'>
              <th>ID</th> <th>Name</th> <th>Email</th> <th>Status</th>
            </tr>

            return JSX;
    }

    render(){
        return (
            <div>
              <table  className="tableStyle" border = '1|1'>
                <tbody>
                    {this.buildHdr()}
                    {this.buildRows()}
                </tbody>
              </table>
            </div>
        );
     }
  }
```

```
import React from 'react';
import "./CSS/tableStyles.css"
export class ToDosPage extends React.Component {
   constructor(props){
      super(props);
       this.buildHdr = this.buildHdr.bind(this);
       this.buildRows = this.buildRows.bind(this);
    }

    buildRows(){
      let I = 0;
      const list = this.props.toDos.map(t => {
        let oneRow = <tr className='rowStyle' key = {I++}>
        <td>{t.userId}</td> <td>{t.item}</td> <td>{t.due}</td> <td>{t.status}</td>
        </tr>

            return oneRow;
        });

        return list;
     }

     buildHdr(){
        let JSX =
         <tr className='rowStyle'>
         <th>User ID</th> <th>Item</th> <th>Due</th> <th>Status</th>
         </tr>
         return JSX;
  }
  render(){
    return (
        <div>
          <table className="tableStyle" border = '1|1'>
            <tbody>
              {this.buildHdr()}
              {this.buildRows()}
            </tbody>
          </table>
        </div>
    );
  }
}
```

Note: the Users Page and To Dos Page components require data via their props, i.e., from their parent component.

- this.props.users
- this.props.toDos

Both components have a VERY SIMPLE style.

import "./CSS/tableStyles.css"

```
.tableStyle {
    width: 67%;
    border: 1px solid black;
    border-collapse: collapse;
    margin-top: 10px;
    margin-bottom: 0px;
    margin-left: 3%;
    margin-right: 0px;
    float: center;
}

.rowStyle {
        border: 1px solid black;
        border-collapse: collapse;
        background-color:lightgrey;
        text-align: center;
}
```

The (incomplete for now!) parent App component

Style

```
.menu{
background-color: lightgray;
list-style-type:none;
padding: 4px;
margin:3px;
width:28%;
}

.menuItem{
    text-decoration: none;
    display: inline-block;
    padding: 8px;
}

.active{
    background-color:lightsalmon;
}

.blink_me {
    animation: blinker 1s linear infinite;
  }

  @keyframes blinker {
    50% {
      opacity: 0;
    }
  }
```

```
import React from 'react';
import {Route, NavLink, HashRouter} from "react-router-dom";
import {Home} from "./home.js"
import {UsersPage} from "./usersPage.js"
import {ToDosPage} from "./toDosPage.js"
import "./CSS/mainMenuStyle.css"
import { EntityDataFetch }     from './entityDataFetch.js';
import {User} from "./entities/user.js"
import {ToDoItem} from "./entities/toDoItem.js"
import {EntityFetchPacket} from './entityFetchPacket.js';

export class App extends React.Component{

    usersUrl ='https://gorest.co.in/public/v1/users';
    usersJsonKey = 'data';

    todosUrl ='https://gorest.co.in/public/v1/todos';
    todosJsonKey = 'data';

    constructor(props){
        super(props);
        this.users = [];
        this.todos = [];

        this.state = {dataLoadComplete: 'PENDING'};
        this.dataLoadedHandler = this.dataLoadedHandler.bind(this);

        this.fetchIssue = null;
        this.handleFetchException = this.handleFetchException.bind(this);

        this.dataLoadingRender = this.dataLoadingRender.bind(this);
        this.dataRender = this.dataRender.bind(this);

        this.dataLoadedCount = 0;
    }

    dataLoadedHandler(){
        this.dataLoadedCount ++;
        if (this.dataLoadedCount === 2){
            this.setState({dataLoadComplete : 'COMPLETE'});
        }
    }
```

```
handleFetchException(e){
    this.fetchIssue = e;
    this.setState({dataLoadComplete : 'ERROR'});
}

componentDidMount(){  // for NOW: simulate two arrivals users and todos
    this.dataLoadedHandler();
    this.dataLoadedHandler();
}

render(){
    console.log ("render");
    let JSX;

    if (this.state.dataLoadComplete === 'COMPLETE'){
        JSX = this.dataRender()
    } else if (this.state.dataLoadComplete === 'PENDING'){
        JSX = this.dataLoadingRender()
    } else {
        JSX = this.errorRender()
    }

    return JSX;
}

dataLoadingRender(){
   return (
   <div>
   <h1>Welcome to Cold Beans for lunch Ink</h1>
   <p className="blink_me">Please be patient.  We are initilaizing data...</p>
   <Home></Home>
   </div> );
}

errorRender(){
    let errorMsg = 'Loading Data Error occurred Call IT now! Provide the msg: '
          + this.fetchIssue;

   return <div>{errorMsg}</div>
}
```

```
dataRender(){
        let users = this.users;
        let todos = this.todos;
        var JSX =
        <HashRouter>
        <div>
           <h1>Welcome to Cold Beans for lunch Ink</h1>

           <p>Menu Options</p>
           <ul className="menu">
           <li className="menuItem"><NavLink exact to= "/">Home</NavLink></li>
           <li className="menuItem"><NavLink to= "/users">Users</NavLink></li>
           <li className="menuItem"><NavLink to= "/toDosPage">To Dos</NavLink></li>

           </ul>

           <div>
           <Route exact path= "/"  component= {Home}></Route>
           <Route path= "/users" render={(props) =>
              <UsersPage {...props} users={users} />}></Route>
           <Route path= "/toDoListList" render={(props) =>
              <ToDosPage {...props} toDos={todos} />}></Route>
           </div>
        </div>
        </HashRouter>

        return JSX;
    };
};
```

Notes

- We have NO data to display yet!
- Our children require User and To Do related data sets
- The componentDidMount function simulates the two data sets arriving with no data.

We now add some test data.

At the bottom of the constructor add:

```
this.testData = this.testData.bind(this);
```

Change did mount function. Test data is loaded and we still ‘fake’ two data arrivals.

```
componentDidMount(){

    this.testData();

    this.dataLoadedHandler();
    this.dataLoadedHandler();
  }
```

Add the new function.

```
testData(){
        let userObj =
        {id: 2222, name: 'Nadia', email:'gaga@gmail.com', status:'Active'};
        let user = new User(userObj);
        this.users.push(user);

        userObj =
        {id: 2222, name: 'Hope', email:'hopesmail.com', status:'On Leave'};
        user = new User(userObj);
        this.users.push(user);

        let toDoObj =
        {user_id: 2222, title: 'Go to Bank', due_on:'Jan 12 2021', status:'Completed'};
        let toDo = new ToDoItem(toDoObj);
        this.todos.push(toDo);

        toDoObj =
        {user_id: 2222, title: 'Party at Baileys ', due_on:'Sept 12 2021', status:'Pending'};
        toDo = new ToDoItem(toDoObj);
        this.todos.push(toDo);

        toDoObj =
        {user_id: 2222, title: 'Study React ', due_on:'Sept 12 2021', status:'In Progress'};
        toDo = new ToDoItem(toDoObj);
        this.todos.push(toDo);
    }
```

Run the application. You should see an initial quick screen '**blink'.** Why? How many times to we render? Click here and there…

Remote data

Now we remove the test data and use real remote data via these public URLs. Try them out via Postman

- https://gorest.co.in/public/v1/todos
- https://gorest.co.in/public/v1/users

Remove **all** the test data related code. Suggestion, copy and rename the App component first.

We have two entities: User and To Do Item.

For a remote fetch we will need the data.

```
export class EntityFetchPacket{
   url;
   jsonKey;
   entityArr;
   entityClass;
   errorHandler;
   dataLoadedHandler;
}
```

1. The remote URL
2. The JSON array name. The remote server returns an JSON array of data. The key is the name of the array. Via Postman verify that both URLs use the key 'data'
3. The entity array is our java script array to hold the Entities.
4. An Entity class: one of: User, To Do Item
5. Error handler for fetch related issues.
6. Data loaded handler called when the remote data (JSON) finally arrives,

We place the java script **fetch** related code in its own class. This class requires **no** knowledge about our two application entities.

```
export function fetchText(url, entityDataFetch) {

  get(url).then (
        (responseText) => {
            entityDataFetch.textHandler(responseText);
        }
        ).catch((e) => {
            entityDataFetch.errorHandler(e);
        })
}

async function  get (url)  {
    let responsePromise = await fetch(url);
    if (!responsePromise.ok){
        let issue = 'ERROR: HTTP issue. Status: ' + responsePromise.status;
        throw new Error(issue);
    }

    let retText = await responsePromise.text();
    return retText;
}
```

This class invokes the above **fetch Text** function and populates the packet's **entity array**.

```
import {fetchText} from './fetchText.js';

export class EntityDataFetch {

    constructor(fetchPacket){
        this.packet = fetchPacket;
    }

    errorHandler(e){
        this.packet.errorHandler(e);
    }

    fetchData(){
        fetchText( this.packet.url, this);
    }

    textHandler (responseText){
        let responseObj = JSON.parse(responseText);
        let rows = responseObj[this.packet.jsonKey];
        let entityClass = this.packet.entityClass;

        for (var i = 0; i < rows.length; i++) {
            let entity = new entityClass(rows[i]);
            this.packet.entityArr.push(entity);
        }

        // our entity array is all loaded, so…
        this.packet.dataLoadedHandler();
    }
}
```

A note on our ‘packet’ class via an example:

Style 1

```
function xxx(lname, fname, minitial, address, zip){
   in here there is logic that requires the FIVE string arguments
}
```

Invoker: xxx(‘James’, ‘K’, ‘Kelly’, ‘109888’, ‘23 North No Place Ave’)

This compiles and runs but the invoker must be a ‘drunken sailor’!

Style 2

```
Function xxx(xxxPacket){
   in here there is logic that requires one argument
}
```

Invoker:

```
packet = new xxxPacket()
xxx.fName = ‘Kelly’
xxx.lName = ‘James’
xxx.minitial = ‘K’
xxx.zip = ‘109888’
xxx.address = ‘23 North No Place Ave’

xxx(packet){
…..
}
```

I prefer **style 2** especially since java script is **NOT** typed. One may say that it is a matter of opinion. If so, and you do not share my opinion then change to **style 1.** I will not be offended; ok, may a little offended.

The parent App

```
import React from 'react';
import {Route, NavLink, HashRouter} from "react-router-dom";
import {Home} from "./home.js"
import {UsersPage} from "./usersPage.js"
import {ToDosPage} from "./toDosPage.js"
import "./CSS/mainMenuStyle.css"
import { EntityDataFetch }     from './entityDataFetch.js';
import {User} from "./entities/user.js"
import {ToDoItem} from "./entities/toDoItem.js"
import {EntityFetchPacket} from './entityFetchPacket.js';

export class App extends React.Component{

    usersUrl ='https://gorest.co.in/public/v1/users';
    usersJsonKey = 'data';

    todosUrl ='https://gorest.co.in/public/v1/todos';
    todosJsonKey = 'data';

    constructor(props){
        super(props);
        this.users = [];
        this.todos = [];

        this.state = {dataLoadStatus: 'PENDING'};
        this.dataLoadedHandler = this.dataLoadedHandler.bind(this);

        this.fetchIssue = null;
        this.handleFetchException = this.handleFetchException.bind(this);

        this.dataLoadingRender = this.dataLoadingRender.bind(this);
        this.dataRender = this.dataRender.bind(this);

        this.dataLoadedCount = 0;
    }
```

The **state** keeps track of the load status, initially **PENDING**. The state changes to either **COMPLETE** or **ERROR** depending how our two fetches turn out.

These two functions handle the two possible fetch outcomes: **COMPLETE** or **ERROR.** Both functions perform a state change, i.e., (a re-render)

```
dataLoadedHandler(){
      this.dataLoadedCount ++;
      if (this.dataLoadedCount === 2){
          this.setState({dataLoadStatus : 'COMPLETE'});
      }
   }

 handleFetchException(e){
      this.fetchIssue = e;
      this.setState({dataLoadStatus : 'ERROR'});
   }
```

Our component Did Mount function should return immediately. The actual remote data arrives some time afterward.

```
componentDidMount(){
     const  usersPacket = new EntityFetchPacket();
     usersPacket.url = this.usersUrl;
     usersPacket.jsonKey = this.usersJsonKey;
     usersPacket.entityArr = this.users
     usersPacket.entityClass =User;
     usersPacket.errorHandler = this.handleFetchException;
     usersPacket.dataLoadedHandler = this.dataLoadedHandler;
     const userDataFetch = new EntityDataFetch(usersPacket);
     userDataFetch.fetchData();

     const  todosPacket = new EntityFetchPacket();
     todosPacket.url = this.todosUrl;
     todosPacket.jsonKey = this.todosJsonKey;
     todosPacket.entityArr = this.todos
     todosPacket.entityClass =ToDoItem;
     todosPacket.errorHandler = this.handleFetchException;
     todosPacket.dataLoadedHandler = this.dataLoadedHandler;
     const todosDataFetch = new EntityDataFetch(todosPacket);
     todosDataFetch.fetchData();
   }
```

What is rendered depends on our **state's load status**

```
render(){
      let JSX;

      if (this.state.dataLoadStatus === 'COMPLETE'){
          JSX = this.dataRender()
      } else if (this.state.dataLoadStatus === 'PENDING'){
          JSX = this.dataLoadingRender()
      } else {
          JSX = this.errorRender()
      }

      return JSX;
  }
```

We have three JSX **builders**; one for each our **state's load status**

```
dataLoadingRender(){
    return (
    <div>
    <h1>Welcome to Cold Beans for lunch Ink</h1>
    <p className="blink_me">Please be patient.  We are initilaizing data...</p>
    <Home></Home>
    </div> );
  }

  errorRender(){
     let errorMsg =
           'Loading Data Error occurred Call IT now! Provide the msg: ' + this.fetchIssue;
     return <div>{errorMsg}</div>
  }

  dataRender(){
     let users = this.users;
     let todos = this.todos;
     var JSX =
     <HashRouter>
     <div>
        <h1>Welcome to Cold Beans for lunch Ink</h1>
        <p>Menu Options</p>
        <ul className="menu">
        <li className="menuItem"><NavLink exact to= "/">Home</NavLink></li>
        <li className="menuItem"><NavLink to= "/users">Users</NavLink></li>
        <li className="menuItem"><NavLink to= "/toDos">To Dos</NavLink></li>
        </ul>
        <div>
        <Route exact path= "/"  component= {Home}></Route>
        <Route path= "/users" render={(props) =>
           <UsersPage {...props} users={users} />}></Route>
        <Route path= "/toDos" render={(props) =>
           <ToDosPage {...props} toDos={todos} />}></Route>
        </div>
     </div>
     </HashRouter>

     return JSX;
  };
```

Run the application. If you do not see data check the two URLs in Postman. If the data appears in Postman start your debugging!

Slow Remote Servers

When I run the application the remote data arrives very quickly. I want to slow things down a bit and look at the application while it is in the **PENDING** state**.**

Back to the fetch Text class to insert a 'fake' slowdown. The code follows.

```
export function fetchText(url, entityDataFetch) {

    fakeWait(10000).then(() => {
        get(url).then (
            (responseText) => {
                entityDataFetch.textHandler(responseText);
            }
            ).catch((e) => {
                entityDataFetch.errorHandler(e);
            })
        }
        )
    }

async function  get (url)  {
    let responsePromise = await fetch(url);
    if (!responsePromise.ok){
        let issue = 'ERROR: HTTP issue. Status: ' + responsePromise.status;
        throw new Error(issue);
    }

    let retText = await responsePromise.text();
    return retText;
}

async function  fakeWait (ms)  {
    const myPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
          resolve('all done!');
        }, ms);
      });

    return myPromise;
}
```

Note: the fake delay is **10** seconds: fakeWait(10000). Time an application run and note the time it takes before the re-rendering with the remote data. We do **two** fetches. Make sure you understand why the application does **NOT** delay **20** seconds

Things for you to look at.

Scrolling: the number of users and/or to do items may be too large to fit on your page

What if there is a to do item with NO corresponding user. Is it your React application's responsibility to deal with this issue?

Suppose the Users page needs/wants access to the parent's **array** of to do items. The parent can include a simple getter **function** that returns a reference to the **array** and send a reference to the **function** in the Users page component. Example use: When the application web-user clicks on a user the corresponding list of to do items appears!

…and so on…

Some very informal definitions used in the paper.

Fetch API. The java script Fetch API for POSTs and GETs etc.

HOC (higher order component) is a java script function that takes a component definition as an input and returns another component definition.

JSX (JavaScript XML) An HTML like markup that React converts to HTML

Life cycle. A component's life cycle.

MVC: the Model View Control architecture

Mount: Once a React component is fully initially instantiated it is said to be 'mounted'

Packet : A term used in this paper. The technique of converted a function with multiple arguments to a function with one argument

REST (Representational State Transfer): a software architecture - in this context, a set of guidelines for creating stateless WEB based API

React Class component. A java script class that extends react Component. The basic building block of a React application

React Functional Component. A java script function that returns JSX. React can render the JSX for us!

Reference: in React, a reference to a DOM element

Render prop: a java script function passed to a child component via the 'props' object to assist the child's render logic

SPA single page application. A React mechanism for creating a single page WEB application that appears to the web user as a multiple page application

State: the React Component's state java script object that every user defined class component inherits

Synthetic Event a java script class used by React. The class wraps a DOM event.

Update: When a component is rendered after the initial render. A render invocation after the component is mounted

Use State Hook: allows React functional components to have a persistent state (maintained over N function invocations)

Widget whatever you want it to be !

props (lower case): a java script object used to pass data from a parent component to a child component; it can include reference to objects, primitives or functions.

react-router-dom (lower case) : The React lib to facilitate SPA development

The End