The Full React Guide

- *Why Learn React:*
    - Small learning curve, so you can get productive quick! Builds off Javascript and JSX.
    - Community: Easy to get answers when you get stuck! Also, many things already developed so you don't have to develop core aspects.
    - The library itself: Component based architecture makes large apps easier to develop. Components are little pieces that make up the whole app. They reusable and easy to build and debug!
- *Setting up your system for React:*
    - Install Visual Studio Code.
    - Install node.js and yarn.
        1. In your terminal: npm install –g yarn
        2. Restart system
- *Your First React App:*
    - **Indecision App:** An app that you add a list of things to do, and the app picks which one you should do for you.
        1. **Key Learning Points:**
            - JSX: JavaScript XML. Templating language used to build out the user interface for components. JavaScript syntax extension, provided to us by React. Great way to define and inject data into templates.
            - **Babel**: Your browser does not know what to do with JSX. It only works with JavaScript. Babel compiles down our JSX to JavaScript so we can view our site. It alone does nothing, we must add presets (group of plugins) for Babel to work. We need to include the React and env preset. React preset allows us to use JSX inside our code. Env allows us to use es6 and es7 features, like const, arrow functions, and rest and spread operators. Install:
                - 1. npm install -g babel-cli@6.24.1 (gives us command line interface to use babel but does not give us the needed presets)
                - 2. yarn init (generates a file named package.json -> outlines all of the dependencies needed for our project to run. This allows us to quickly install all dependencies using something like npm install at any time)
                - 3. yarn add babel-preset-react@6.24.1 babel-preset-env@1.5.2 (for presets – you will notice a new folder called node_modules, this holds the sub-dependencies for our presets. You will also see a yarn.lock file -> autogenerated file, lists all dependencies in node_modules, and lists where it got these dependencies)
        2. babel src/app.js --out-file=public/scripts/app.js --presets=env,react --watch (sets what file to JSX file should be autogenerated by babel. First dir is our in file,

seconf dir is our generated outfile. --watch can be added on to keep this running in the background so any changes to the in file are automatically converted for you)

- **Live-server:** Bare bones no configuration web server. Allows us to serve up our public folder, and allows us to live refresh.
    1. Install (while in project dir): npm install -g live-server
    2. Run: live-server public

- *File Structure:*
    - App.js in the source folder contains all the JSX we write.
    - App.js in the scripts folder will be autogenerated JavaScript derived from our JSX through babel transformations.
- *JSX Syntax:*
    - All **adjacent tags** must be wrapped in a tag.

```
Ex: var templateTwo = (

    <div>
        <h1> Sanjeev Sharma </h1>
        <p> Age: 25 </p>
        <p> Location: NYC </p>
    </div>
);
```

- **JSX Expressions:** We want our websites to be dynamic and not static. Using Expressions allows us to pull data as variables. This can be useful for population multiple names from databases etc…

```
-   var userName = 'Sanjeev Sharma';
-   var userAge = 25;
-   var userLocation = 'NYC';
-
-   var templateTwo = (
-       <div>
-           <h1>{userName.toUpperCase() + '!'}</h1>
-           <p>Age: {userAge}</p>
-           <p>Location: {userLocation}</p>
-       </div>
-   );
```

1. Notice the use of concatenation and string methods that this allows us as well.
2. **Objects:** All this user data can be defined as an object:

```
-   var user = {
-       name: 'Sanjeev Sharma',
-       age: '25',
-       location: 'NYC',
-   };
-   var templateTwo = (
```

```
        <div>
            <h1>{user.name.toUpperCase() + '!'}</h1>
            <p>Age: {user.age}</p>
            <p>Location: {user.location}</p>
        </div>
    );
```

1. **Conditional Rendering in JSX:** What if something we want to render is not populated yet? For this we can use conditional statements. Issue with conditional statements is that they cannot exist where JavaScript expressions are. Instead you must make a function, and call that function with the variable as a parameter (highlighted in red):

```
    function getLocation(location){
        if (location) { // if location does exist
            return location;
        }
        else{
            return 'Unknown';
        }
    }
    var templateTwo = (
        <div>
            <h1>{user.userName}</h1>
            <p>Age: {user.userAge}</p>
            <p>Location: {getLocation(user.userLocation)}</p>
        </div>
    );
```

1. Or if you want nothing to display at all in the case the variable had not populated:

```
    function getLocation(location){
        if (location) { // if location does exist
            return <p>Location: {location} </p>;
        }
    }
    var templateTwo = (
        <div>
            <h1>{user.userName}</h1>
            <p>Age: {user.userAge}</p>
            {getLocation(user.userLocation)}
        </div>
    );
```

1. We put the paragraph tag in the function and simply call the function in curly braces in the div. Notice the lack of else statement means that if one of our variables shows up as undefined, nothing will show on our app for that variable.

2. **Ternary Operator:** More concise than creating a function. No need to break out to separate function, you can do this inline. This is because it is an expression and not a statement. Good for if you want to do 1 of 2 things.

```
-   var templateTwo = (
-       <div>
-           <h1>{user.userName ? user.userName : 'Anonymous'}</h1>
-           <p>Age: {user.userAge}</p>
-           {getLocation(user.userLocation)}
-       </div>
-   );
-
```

1. User.userName ? user.userName -> if username exists, return username: 'Anonymous'; -> else return static Anonymous
2. **Logical Operators:** Undefined Booleans are ignored by JSX, which can be very useful. For example, if we only want to display the age of users who are 18 or older we can use the **and** operator:

```
-   var templateTwo = (
-       <div>
-           <h1>{user.userName ? user.userName : 'Anonymous'}</h1>
-           {user.userAge >= 18 && <p>Age: {user.userAge}</p>}
-           {getLocation(user.userLocation)}
-       </div>
-   );
```

1. if the first part of the and statement is true, the second part is returned and shown. If it is false, false is returned, and as we learned, undefined Booleans are ignored so nothing will show. This is exactly what we want! It is good for if you want to do 1 thing or nothing at all.
2. You can also check if age exists by nesting an and statement to check for it:

```
-   {(user.userAge && user.userAge >= 18) && <p>Age: {user.userAge}</p>}
```

- *ES6:*
    - **let, const**
        1. let: Issue with using var is that it is redefinable with no errors. There is no useful case for this and can cause problems. let on the other hand is not redefinable and throws an error in your terminal. You can always reassign let variables, but it is not redefinable.
        2. const: like let, this is not redefinable. But since this is a constant variable, it is also not reassign able.
        3. Scoping: var, let, const are all function scoped. let and const are also block scoped. This means that these variables are not only unique to their functions and cannot be accessed from outside the function, they also cannot be accessed outside of their block. Block scoping means if you define a variable in something like a for loop or if statement, these variables are unique to these blocks and cannot be accessed from outside of this scope.

- **Arrow Functions:** A brand new syntax for creating functions offered through es6.

```
const squareArrow = (x) => {
    return x*x;
};
console.log(squareArrow(10));
```

1. Notice that the function name is now anonymous, so you cannot define a function by name and you must use a variable.
2. *Expression Syntax*: allows us to be more concise with our functions by not having a function body. Expression syntax functions do not have a return, instead the single expression is implicitly returned. Good for functions that return a single expression:

```
const squareArrowExp = (x) => x * x;
console.log(squareArrowExp(11));
```

1. You cannot use **arguments** with arrow functions:

```
const addArrow = (a, b) => {
    console.log(arguments);
    return a + b;
};
```

1. This does not work as arguments are no longer bounded using arrow functions. Use ES5 functions if you must access arguments.
2. **this** is only works for functions that are object properties in ES5, and not for random anonymous functions:

```
const user = { //ES5
    name: 'Sanjeev',
    cities: ['NYC', 'Queens', 'Miami'],
    printPlaceLived: function(){ // since this function is added to the object property, the this is bound to that object
        console.log(this.name);
        console.log(this.cities);

        this.cities.forEach(function (city) {
            console.log(this.name + ' has lived n' + city); // does not work since this anonymous function is not bound to the object
        });
    }
};

```

1. On the other hand, arrow functions inherit parents this. They use the this value of the context they were created in:

```
const user1 = {
    name1: "Sanjeev1",
    cities1: ['NYC', 'Queens', 'Miami'],
    printPlaceLived(){ // ES6 syntax for defining a method function
```

```
-           this.cities1.forEach((city) => {
-               console.log(this.name1 + 'has lived in ' + city); // ES6 works since arrow functions inherit the this of the context they are created
-           });
-       }
-   };
```

1. **Map:** allows you to transform each item in an array and returns it in a new array.

```
-   const user2 = { //using map
-       name2: 'Sanjeev2',
-       cities2: ['NYC', 'Queens', 'Miami'],
-       printPlaceLived(){
-           return this.cities2.map((city)=> this.name2 + ' has lived in ' + city); //map allow you to transform each item in the arry and get a new array back
-       }
-   };
```

- *JSX Attributes:* While most html attributes carry over to JSX, there are some key differences in attributes:
    - class attribute is used to add identifiers to elements. These identifiers can be shared across multiple elements. This is good for styling using something like bootstrap. Class is now reserved for class declaration in JSX, so instead we use className:

```
-   let count = 0;
-   const templateTwo = (
-       <div>
-           <h1> Count: {count} </h1>
-           <button id ="my-id" className="button"> +1 </button>
-       </div>
-   );
-
```

    - Some of the HTML attributes that do carry over are now camel cased instead of being all lower case. You can use https://reactjs.org/docs/dom-elements.html for reference.
- *Events:* You want to be able to program dynamic changes to data from user. This can be done through events. To set up events, you can create functions:

```
-   let count1 = 10;
-   let initial = count1;
-   const minus = () => console.log('minus');
-   const reset = () => console.log('reset');
-   const templateThree = (
-       <div>
-           <h1> Count: {count1} </h1>
-           <button onClick= {add1} className="button"> +1 </button>
-           <button onClick = {minus} className="button"> -1 </button>
-           <button onClick = {reset} className="button"> reset </button>
```

```
-       </div>
-   );
```

- **button onClick:** you can either call a function onClick or you can even inline some function. It is always better to pull out functions though if you will be needing the function in multiple places in code.
- *Manual Data Binding:* In the last snippets of code we do not re-render our count, but instead console.log what we are doing. JSX does not have built in data binding. This is because templateThree runs before anything is rendered to the screen. This is because nothing renders until ReactDOM.render is called, meaning that whatever are the initial values we declare will be rendered on screen. We can fix this by wrapping templateThree + ReactDOM.render inside a render function and calling this function wherever we need to initially render and where we need it to re-render: (This is called real time manual data binding)

```
-   const reset = () => {
-       count1 = initial;
-       renderCounterApp(); // re-render when function is called
-   };
-
-   const renderCounterApp = () =>{
-       const templateThree = (
-           <div>
-               <h1> Count: {count1} </h1>
-               <button onClick= {add2} className="button"> +1 </button>
-               <button onClick = {minus} className="button"> -1 </button>
-               <button onClick = {reset} className="button"> reset </button>
-           </div>
-       );
-       ReactDOM.render(templateThree, appRoot);
-   };
-   const appRoot = document.getElementById('app');
-   renderCounterApp(); // initial call to render
```

- This may look horribly inefficient as it looks like our web app is re-rendering the whole templateThree just to change a small element of it, React runs a virtual DOM algorithm that takes this and re-renders only the minimal things that are needed.

- *Forms and Inputs:* How do we handle forms and user input on those forms? We set up a form tag and a input tag:

```
-               <form onSubmit={onFormSubmit}> {/* reference the function, do
    not call it.*/ }
-                   <input type="text" name="option"/>
-                   <button> Add options </button>
-               </form>
```

- We use form's onSubmit when working with forms, not the submission buttons onClick. We want to watch for the whole form to submit.
- Now we set up the custom event onSubmit handler:

```
- const onFormSubmit = (e) => { // e = event object: contains various
  information about the event object
-     e.preventDefault(); // stops full page refresh
-
-     const option = e.target.elements.option.value; // grab user submitted
  option from form
-     if(option){ // check if option is populated
-         app.options.push(option); // push to options vector
-         e.target.elements.option.value = ''; // empty form text field
-         renderApp(); // re-render app
-     }
- };
```

- Read comments for details on what is happening here. The last call to renderApp() is to re-render our app. Look at Manual Data binding to refresh on this if you must.

- *Arrays in JSX:* JSX supports arrays. You can make inline arrays:

```
-             <p> {app.options.length} </p>
-             {[12,14,13]} // inline array
-             <button onClick = {clearList} className = "button"> Remove All
  </button>
```

- You can even do arrays of JSX in JSX to display a list of array members, this allows us to render a list of dynamic number of items:

```
-             <ol>
-                 {
-                     app.options.map((option) => {
-                         return <li key = {option}> {option} </li>
-                     })
-                 }
-             </ol>
```

- You must use unique keys for each JSX item inside a JSX array so React knows what items to re-render.

- *Random Number Generator:* Our app needs to randomly select one item from the users todo list. Logically, this mean that our app needs to randomly pick a index from our options array. This is possible using the Math.random() function. Math.random() returns a decimal number between 0 and .99 so it would not be useful alone. We can fix this by multiplying this number by the size of the array: Math.random() * app.options.length. This makes our new range from 0 to just about the size of our array. Even this is not complete though. You will notice that array indeses are whole numbers. We can use Math.Floor to round down numbers to their neared whole:

```
- const makeDecision = () =>{
-     const randomIndex = Math.floor(Math.random() * app.options.length);
-     console.log(randomIndex);
- };
```

- We can now use this random index to access the data in the array at this index:

```
- const makeDecision = () =>{
-     const randomIndex = Math.floor(Math.random() * app.options.length);
-     const option = app.options[randomIndex];
-     alert(option);
- };
```

- **alert** is used to display a pop up of the randomly selected choice.
- We can tack on a disabled with a jsx function to disable the button when the list is empty:

```
- <button disabled={app.options.length == 0} onClick = {makeDecision}
  className = "button"> Pick4me </button>
```

- We can now call this function as a onClick for new button labeled Pick4Me:

```
- <button onClick = {makeDecision} className = "button"> Pick4me </button>
```

- *Build-It Challenge 1:* Visibility Toggle – Build from scratch a new web app with the header "Visibility Toggle". Include a button that is initially labeled "Show Details". When pressed, the label changes to "Hide Details" and a new paragraph is generated with the following text "Hey. There are some details you can see now! ". Upon a second click to this button, this paragraph goes away:

- Start by setting up manual data binding:

```
- const renderApp = () => {
-     const template = (
-         <div> </div>
-     );
-     ReactDOM.render(template, appRoot);
- };
- const appRoot = document.getElementById('app');
- renderApp();
```

- Set up a class for app attributes:

```
- const app = {
-     title: 'Visibility Toggle',
-     toggle: false,
- };
```

- Set up header and button. Our button uses the ternary operator to label the button one of two things depending on our toggle state. If toggle is true then Hide details, else Show details:

```
-     const template = (
-         <div>
-             <h1> {app.title} </h1>
-             <button onClick = {toggleButton} className = "buttons">
  {app.toggle ? 'Hide Details' : 'Show Details'} </button>
-         </div>
-     );
```

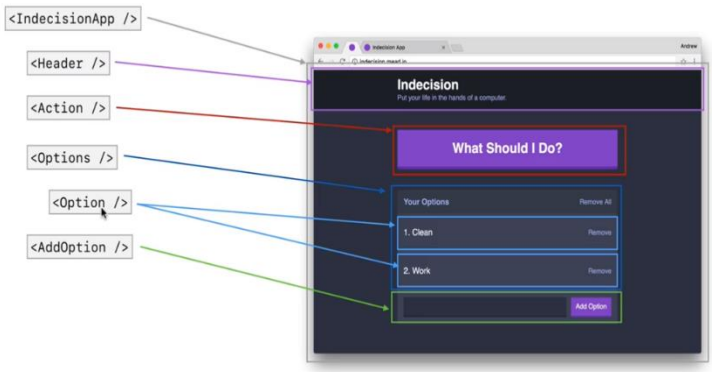
- Set up a function to toggle our bool value when the button is pressed, and call this function for our buttons onClick. Be sure to re-render app!:

```
const toggleButton = () =>{
    if(!app.toggle){
        app.toggle = true;
    }
    else{
        app.toggle = false;
    }
    renderApp();
};
```

- Finally, use the and operator to display or hide our message when button is pressed. This is done by checking our toggle state first. We use the and operator instead of ternary because we want to display 1 message or nothing at all:

```
            {app.toggle && (
                <p> Hey. These are some details you can now see! </p>
            )}
```

- *React Components:* Allows us to break up our application to small reusable chunks. This is good for parts that will appear on multiple pages. Think of something like a menu bar that is always available on every page of a website. You would not want to have to rewrite this menu bar multiple times, so instead we can create a component and reuse it throughout our site. Another example of this is your newsfeed on facebook. While each post may be different, the underlying structure and JSX is the same. This sets up a good scenario for using a React Component.
    - Each component has its own set of JSX that it renders to the screen
    - It can handle events for those JSX elements and allows us to create small self-contained units.
    - You can nest components like HTML. In our case our <Pick4Me /> parent component will have its own child components <Header />, <Action /> for our pick button, <Options />, for our list of options, and <AddOption /> for our add option button. <Options /> will be the parent for <Options /> which will be our individual options.
    - Each parent renders its children, but not its grandchildren. <Options /> will render <Option /> , not <Pick4Me />.



- *ES6 Classes:* Classes are like blueprints to a building. Once you have this blueprint, you can set up multiple of the same building quickly. The blueprint would be considered a declaration, and

each building made would be considered a instance of this blueprint. These building will share common things, like an address. But each address can be different. Classes are simply generic constructs that we create instances for.

- **Constructor() :** The constructor is a function that creates an instance of a class. It can have arguments, or not depending on what your class is. Notice that we can assign a default value to constructor arguments in case with create an instance without passing that variable.

```
class Person {
    constructor(name, location = 'Not Given'/*argument default value*/){
        this.name = name;
        this.location = location;
    }
}
```

- **Class Methods:** Simply a class function. This function has access to class members and variables using (this):

```
class Person {
    constructor(name, location = 'Not Given'/*argument default value*/){
        this.name = name;
        this.location = location;
    }

    getGreeting(){ // class method
        //return 'Hi ' + this.name;
        return `Hi ${this.name}`; // ES6 template string using back ticks
    }
}
```

- getGreeting() is a class method that accesses our class name variable.
- ES6 template strings allow us to access member variables within our strings instead of having to use concatenation of the variable on to the string.
- **Declaring an instance of a class:**

```
const me = new Person('Sanjeev Sharma', 'NYC');
const someone = new Person('Random');
```

- Location for someone will be 'Not Given' since we do not pass a location and our default value is that.

- *Subclasses:* Classes that extend other classes. This allows the child class to get all of the behavior of its parent class without copying any code.

```
class Person2{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
```

```
-       getGreeting(){
-           return `Hi. I am ${this.name}. `;
-       }
-
-   }
-
-   class Traveler extends Person2{
-       constructor(name, age, location){
-           super(name, age);
-           this.location = location;
-       }
-       homeLocation(){
-           return !!this.location;
-       }
-       getGreeting(){
-           let greeting = super.getGreeting();
-           if(this.homeLocation()){
-               return greeting + `I'm visting from ${this.location}.`
-           }
-           return greeting;
-       }
-   }
```

  - Our child class has its own constructor that calls its parents constructor first with super(). Super() is used again to access parent getGreeting() method in the childs getGretting() method.
  - In this case we added a new method to the child class, along with override a method from the parent class. homeLocation() returns false if there is no location, allowing use to override the parent getGreeting() method in the case we do have a location.

- *React Components:* Allow you to split the UI into independent, reusable pieces, and think of each piece in isolation. They are simply es6 classes that extends another class. In this case, they extend a class given to us by react, React.components:

```
-   class Header extends React.Component {
-       render(){// required to be defined in react components
-           return (
-               <div>
-                   <h1> Pick4Me </h1>
-                   <h2> Let Us Choose! </h2>
-               </div>
-           );
-       }
-   }
```

  - React requires you to use uppercase names react components.
  - Every react component must have a render method defined!

- To use these components, we simply provide them inside some JSX:

```
const jsx = (
    <div>
        <Header />
    </div>
)

ReactDOM.render(jsx, document.getElementById("app"));
```

- **Nesting Components:** Essential for creating meaningful react apps. Remember components can render JSX. This means our components can render other components:

```
class Option extends React.Component {
    render () {
        return(
            <div>
                <h1> This is Options Component </h1>
                <Options /> (Options component is nested here)
            </div>
        );
    }
}
class Options extends React.Component{
    render(){
        return(
            <div>
                <p> Option Component here </p>
            </div>
        );
    }
}
```

- We can use this to set up a component for our app that refers to all of the components we created:

```
class Pick4MeApp extends React.Component {
    render() {
        return (
            <div>
                <Header />
                <Action />
                <Option />
                <AddOptions />
            </div>
        );
    }
}
```

- And we can render this main component like so:

```
- ReactDOM.render(<Pick4MeApp />, document.getElementById("app"));
```

- *React Component Props:* Central to what allows our individual components to communicate.

```
- class Pick4MeApp extends React.Component {
-     render() {
-     const t1 = 'Pick4Me';
-     const st1 = 'Let Us Choose!';
-         return (
-             <div>
-                 <Header title = {t1} subtitle = {st1}/>
-                 <Action />
-                 <Option />
-                 <AddOptions />
-             </div>
-         );
-     }
- }
-
- class Header extends React.Component { // must use uppercase for class
  name
-     render(){// required to be defined in react components
-         return (
-             <div>
-                 <h1> {this.props.title} </h1>
-                 <h2> {this.props.subtitle} </h2>
-             </div>
-         );
-     }
- }
```

- We can pass the value that we need for components through props. Props are set up using key value pairs. These look a lot like html attributes. An example of this can be seen above where our <Header /> has a title and subtitle. We can access these props within our component using this.props.nameofkey.

- *Events and Methods:* You can create self containing classes to handle events. Instead of referencing some global method for onClick as we did previously, we simply create a brand new class methods:

```
- class Action extends React.Component {
-     handlePick(){
-         alert('clicked');
-     }
-
-     render(){
-         return (
-             <div>
```

```
                    <button onClick={this.handlePick}> What Should I do?
    </button>
                </div>
            );
        }
    }
```

- You can apply all of your knowledge on JSX event handlers to your class based components:

```
    class AddOptions extends React.Component {
        handleFormSubmit(e){ // add to options vector
            e.preventDefault();
            const option = e.target.elements.option.value.trim(); // .trim()
    removes all leading and ending whitespace.
            console.log(option);
            if(option){
                alert(option);
            }
        }

        render(){
            return (
                <div>
                    <form onSubmit = {this.handleFormSubmit}>
                        <input type = 'text' name = "option"/>
                        <button> Add Option </button>
                    </form>
                </div>
            );
        }
    }
```

- You can apply all of your knowledge on JSX event handlers to your class based components:

- *Method Binding:* You will quickly notice that you do not have access to this members in your component class methods.

```
    class Options extends React.Component {
        handleRemoveAll(){
            console.log(this.props.options); // DOES NOT WORK, OUR THIS
    BINDING IS GONE HERE!
            //alert('removed');
        }
        render () {
            return(
                <div>
```

```
-               <button onClick = {this.handleRemoveAll}> Remove All </button>
-                   {
-                       this.props.options.map((option) => <Option key
    ={option} optionText = {option} />)
-                   }
-                   <Option />
-               </div>
-           );
-       }
-   }
```

-

- You can solve this by using bind:

```
-   <button onClick = {this.handleRemoveAll.bind(this)}> Remove All </button>
```

- But this is not efficient since we would have to run bind every time the component re-renders. A better way to do this is to override the constructor of React.Component:

```
-   class Options extends React.Component {
-       constructor(props){
-           super(props);
-           this.handleRemoveAll = this.handleRemoveAll.bind(this);
-       }
-       handleRemoveAll(){
-           console.log(this.props.options);
-           //alert('removed');
-       }
-       render () {
-           return(
-               <div>
-               <button onClick = {this.handleRemoveAll}> Remove All </button>
-                   {
-                       this.props.options.map((option) => <Option key
    ={option} optionText = {option} />)
-                   }
-                   <Option />
-               </div>
-           );
-       }
-   }
```

- Constructors for react components are called with the props object. We then call super with props. If this is not done, we will not have access to this.props.

- *Component State:* Allows our components to manage some data. Think about an object with various key values. State allows us to automatically re-render our component when we change our values. This contrasts with having to manually re-render as we saw earlier.
  - Steps:
    1. Setup default state object inside your component constructor.

```
-       constructor(props){
-           super(props);
-           this.handleReset = this.handleReset.bind(this);
-           this.handleAddOne = this.handleAddOne.bind(this);
-           this.handleMinusOne = this.handleMinusOne.bind(this);
-
-           this.state = { // Default State Object
-               count: 0,
-           };
-       }
```

2. Component rendered with default state values automatically.

```
    render(){
        return(
            <div>
                <h1> Count: {this.state.count}</h1>
            </div>
        );
    }
```

3. Change state based on event. setState() takes a function as its parameter, and that function takes the prevState object as its own. This allows you to manipulate the data and have it automatically re-render.

```
    handleAddOne(){
        this.setState((prevState) => {
            return{
                count: prevState.count + 1,
            };
        });
    }
```

4. Component re-rendered using new state values automatically
5. Start again at 3 for every change to state

- If the state had more variables other than count, we could access them in this same way. Remember prevState is only the name of an argument. In this case that argument is the state object. Count is a member of this object, as any other variables would also be.

- *Reversing Data Flow –* So far We have seen how we can use state to make changes to our components data. But what if we need to make changes to our parent component from child components? This means passing data to the parent component. This can be done by creating methods in the parent component, and passing these methods as props to the child:

```
-   class Pick4MeApp extends React.Component {
-       constructor(props){
```

```
-           super(props);
-           this.handleRemoveAll = this.handleRemoveAll.bind(this);
-           this.state ={
-               options: [],
-           };
-       }
-       handleRemoveAll() {
-           this.setState(()=>{
-               return{
-                   options: [],
-               };
-           });
-       }
-       render() {
-           const t1 = 'Pick4Me';
-           const st1 = 'Let Us Choose!';
-           return (
-               <div>
-                   <Options
-                       options = {this.state.options}
-                       handleRemoveAll = {this.handleRemoveAll}
-                   />                        ^^^^^^^^^^^^^^^^^^^
-               </div>
-           );
-       }
-   }
```

- And from the child:

```
-   class Options extends React.Component {
-       render () {
-       return(
-        <div>
-         <button onClick = {this.props.handleRemoveAll}> Remove All </button>
-         {                    ^^^^^^^^^^^^^^^^^^^^^^^^^^
-           this.props.options.map((option) => <Option key ={option}
-           optionText = {option} />)
-         }
-         </div>
-           );
-       }
-   }
-
```

- In some cases, we may want to pass an argument to the parent method from a child component. In the parent component:

```
-       handleAddOption(option){
-           this.setState((prevState)=>{
-               return{
-                   options: prevState.options.concat(option),
-               };
-           });
-       }
```

- Notice that we define this method with an argument for option. This option will come from the child component form and added to an array of options. In the parent components render function:

```
-       render() {
-           return (
-               <div>
-                   <AddOptions
-                   handleAddOption = {this.handleAddOption}
-                   />
-               </div>
-           );
-       }
-   }
```

- Now in the child component:

```
-   class AddOptions extends React.Component {
-       constructor(props){
-           super(props);
-           this.handleFormSubmit = this.handleFormSubmit.bind(this);
-       }
-       handleFormSubmit(e){ // add to options vector
-           e.preventDefault();
-           const option = e.target.elements.option.value.trim();
-           this.props.handleAddOption(option);
-       }
-       render(){
-           return (
-               <div>
-                   <form onSubmit = {this.handleFormSubmit}>
-                       <input type = 'text' name = "option"/>
-                       <button> Add Option </button>
-                   </form>
-               </div>
-           );
-       }
-   }
```

- Here in the child component we use the handFormSubmit method to capture our form submission string and pass it to the parent handleAddOption located in the child's props.

- *Props vs. States:*
  - Props are passed from parent component to child component only.
  - Components have access to both props and states while rendering.

Props	State
- An Object	- An Object
- Can be used when rendering	- Can be used when rendering
- Changes from parent component cause re-renders in child component	- Changes cause re-render
- Comes from parent component	- Defined in component itself
- Cannot be changed by child component	- Can be changed by child component using this.setState

- *Stateless functional components:* Alternative to class-based components. Used for simpler components that do not manage state.

```
class StatefulUser extends React.Component{
    render(){
        return(
            <div>
                <p> Name: {this.props.name} </p>
                <p> Age: {this.props.age} </p>
            </div>
        );
    }
}
```

- Good class-based component that can be done as a stateless functional component as this component does not handle state in any way.

```
Const StatelessUser = (props) => {
    return(
        <div>
            <p> Name: {props.name} </p>
            <p> Age: {props.age} </p>
        </div>
    );
}
```

- Notice we no longer need render(), we simply return our jsx
- While state is no longer available here, props are. Remember that props are simply a object. By passing the props object into our stateless functional component, we have access to the props through props.{name of prop}. We no longer need to use this.props.

  - Stateless functional components can then be used just like class-based components. <StatelessUser \>. Props are passed in the same way as class-based components.

    - Stateless functional components are simpler to code, and are faster than class based components as we are no extending the React.Component class. They are also much easier to test.
- *Default Prop Values:* What if a prop is not passed in? We can setup a default value:
    - Use nameOfComponent.deafaultProps:

```
const Header = (props)=>{
    return(
        <div>
            <h1> {props.title} </h1>
            {props.subtitle && <h2> {props.subtitle} </h2>}
        </div>
    );
}
Header.defaultProps = {
    title: 'Pick4Me',
};
```

    - Now if we do not pass in a title prop at the parent component, it will automatically default the title prop to Pick4Me.
    - The same can be done with the main component. You can use this to pass different props for different profiles:

```
class Pick4MeApp extends React.Component {
    constructor(props){
        super(props);
        this.handleRemoveAll = this.handleRemoveAll.bind(this);
        this.handleAction = this.handleAction.bind(this);
        this.handleAddOption = this.handleAddOption.bind(this);
        this.state ={
            options: props.options, // set state to props.option
        };
    }…
Pick4MeApp.defaultProps = {
    options: [], // set default options array to empty
};

```

    - Now if you want to render options on startup for some user just pass in options prop on render:

```
ReactDOM.render(<Pick4MeApp options = {['someOptions1','someOptions2']}/>,
document.getElementById("app"));
```

- *Implicitly Returning an Object:* We can simplify our calls to setStates through the use of implicit arrow functions:

```
    handleRemoveAll() {
        this.setState(()=>{
            return{
                options: [],
```

```
-               };
-           });
-       }
```

- This can be converted to:

```
-       handleRemoveAll() {this.setState(() => ({options: []}))}
```

- Notice that objects get wrapped in parenthesis (in red) since curly braces alone would throw an error since we would not know if this is an object or the function body.

- *Passing Props Multiple Layers Down Components:* This is a terrible title but you will find yourself in situations where you must pass component methods multiple components down. An example of this can be found in my Pick4Me app:
  - We must access a method in our parent component to remove a single option from our option array. The issue is that we have access to our Options (plural) component from the parent component:

```
-       render() {
-       const st1 = 'Let Us Choose!';
-           return (
-               <div>
-                   <Header subtitle = {st1}
-                   />
-                   <Action
-                       hasOptions = {this.state.options.length > 0}
-                       handleAction = {this.handleAction}
-                   />
-                   <Options // ONLY ACCESS TO OPTIONS
-                   options = {this.state.options}
-                   handleRemoveAll = {this.handleRemoveAll}
-                   />
-                   <AddOptions
-                   handleAddOption = {this.handleAddOption}
-                   />
-               </div>
-           );
-       }
```

  - From our Options component we have access to our Option (singular) component. This is where we need to create a button for each individual option:

```
-   const Options = (props) => {
-       return(
-           <div>
-               {
-                   props.options.map((option) => <Option key ={option}
    optionText = {option}/>) // Access to Option Component inside Options
-               }
-           </div>
-       );
```

- We can simply pass the prop from the parent to the Options child:

```
-                   <Options
-                   options = {this.state.options}
-                   handleRemoveAll = {this.handleRemoveAll}
-                   handleRemoveOneOption = {this.handleRemoveOneOption}
-                   /> // PASSING handleRemoveOneCaption as a Prop inside
    parent
```

- And then to Options:

```
props.options.map((option) => <Option key ={option} optionText = {option}
handleRemoveOneOption = {props.handleRemoveOneOption} />)
//Passing the prop further down from Options to Option inside Options component
```

- *Removing Individual Elements From Array:* You can search for and remove individual elements from an array using filter((ObjectToRemove) =>). Filter takes in a function that either returns true or false for each element in the array. This function takes in a parameter which is the item we are looking for. A true return does nothing to the item, while a false return removes the item. So if we wanted to remove some element from the array:

```
-       handleRemoveOneOption(optionToRemove){
-           this.setState((prevState) => ({
-               options: prevState.options.filter((option)=> {
-                   return optionToRemove !== option; // if optionToRemove is
    found, false is returned and filter removes this item and everything else
    stays
-               })
-           }));
-       }
```

- This handleRemoveOnOption is called in the Option component, where we setup a button to call it onClick:

```
-   const Option = (props) => {
-       return (
-           <div>
-               {props.optionText}
-               <button
-                   onClick = {(e) => {
-                       props.handleRemoveOneOption(props.optionText);
-                   }}
-               > Remove </button>
-           </div>
-       );
-   }
```

- We do not directly set onClick to props.handleRemoveOneOption(props.optionText) here because that would call the method immediately. Instead we want to pass a reference to

this prop method. To do this, we inline an arrow function that takes e as an argument so that it can call the function itself when the button is pressed.

- *Lifecycle Methods:* These methods fire at various parts of our app lifecycle. They are not available to stateless components, so if you find yourself needing to use them you must use class-based components. These allow us to watch for changes to state, and save these changes to a database. The following are common lifecycle methods:

```
componentDidMount(){
    console.log('ComponentDidMount');
}
```

  - componentDidMount gets fired when your app first mounts to the DOM. We can fetch data from a database here.

```
}
componentDidUpdate() {
    console.log('ComponentDidUpdate');
}
```

  - componentfDidupdate gets fired whenever there is a change to props or state. This can be especially useful because we can save to a database every time a change occurs to state. We will explore this in relation to a database. We have access to this.state and this.props inside of this method. We can also access previous states and props through arguments to the method. This is useful in the case we do not always want to save something in state, but only when it changes. A quick comparison to this.state and prevState takes care of this.

```
componentWillUnmount(){
    console.log('ComponentWillUnmount');
}
```

  - componentWillUnmount is fired just before your component is exited. This can be useful in sites with multiple pages, where you must clean up before switching pages.

- *JSON and localStorage:* We can explore data persistence through local storage. Our lifecycle methods allows us to save and load data from a local database using JSON stringify and JSON parse:

```
componentDidUpdate(prevState, prevProps) {
    if(prevState.options.length !== this.state.options.length){
        const json = JSON.stringify(this.state.options);
        localStorage.setItem('options',json);
    }
}
```

  - In componentDidUpdate, we only want to save data if there is an actual change to our options array. To check for changes we compare the size of the old state to the new state. If there is a change observed we continue, if there is not we do nothing.
  - We use JSON.stringify(this.state.options) to create a JSON file to be used with localStorage. We then use localStorage.setItem('KeyName', jsonFile) to save this data to the localStorage. The first argument to the setItem method is the kay name.

```
componentDidMount(){
    try{
        const json = localStorage.getItem('options');
        const options = JSON.parse(json);
```

```
-                if(options){
-                    this.setState(()=> ({options: options}))
-                }
-            } catch(e){
-                //do  nothing
-            }
-        }
```

- In componentDidMount we want to retrieve the json that was previously saved in componentDidUpdate, parse it and set our state to this if it exists.
- We use try catch here so that if there is some error in our JSON, our app defaults to an empty options array instead of crashing.
- localStorage stores everything as strings. This means that if you save an int to localStorage, you will have to convert the string back to an int before do anything with it. This can be done with parseInt(nameOfString, base):

```
-        componentDidMount(){
-            const stringCount = localStorage.getItem('count');
-            const count = parseInt(stringCount, 10);
-
-            if(!isNaN(count)){ // check to see if count is a int at this point
-                this.setState(()=> ({count: count}))
-            }
-        }
-        componentDidUpdate(prevProp, prevSate){
-            //console.log('componentDidUpdate');
-            if(prevSate.count !== this.state.count){
-                localStorage.setItem('count', this.state.count);
-            }
-        }
```

- If we tried to convert someString = ‘abc’ to an int, parseInt(someString, 10) returns NaN. We can use this in conjunction with isNaN to check to see if our count is an int before setting out state count to it.