

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.
 - In your terminal: `npm install -g yarn`
 - 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.
 - **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)
 - `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.
 - Install (while in project dir): `npm install -g live-server`
 - 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>  
- );
```

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

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

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

- **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>
- );
```

- 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>
- );
```

- 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.

- **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>
- );
```

- User.userName ? user.userName -> if username exists, return username: 'Anonymous'; -> else return static Anonymous
- **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>
- );
```

- 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.
- 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**

- **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.
- **const:** like let, this is not redefinable. But since this is a constant variable, it is also not reassign able.
- **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));
```

- Notice that the function name is now anonymous, so you cannot define a function by name and you must use a variable.
- *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));
```

- You cannot use **arguments** with arrow functions:

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

- This does not work as arguments are no longer bounded using arrow functions. Use ES5 functions if you must access arguments.
- **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  
-       });  
-   }  
- };  
-
```

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

- **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 array 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>
- );
```

```
-     </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 <Option /> which will be our individual options.
- Each parent renders its children, but not its grandchildren. <Options /> will render <Option />, not <Pick4Me />.

