**React Js for beginners (just to get you started)**

Is react a framework or a library? Well, don’t bother. Reactjs.org calls it a library (collection of precompiled objects based on a programming language, in general) so let it be that while the world fights the third world war over the differences between a library and a framework.

What is the difference between React Js and React Native?

React Js is for web/ browser apps while React Native is for mobile apps.

*Pre-requisite for this tutorial is a basic understanding of HTML, CSS and Javascript.*

The simplest way to get React installed is through Node js (a runtime/ environment\* that allows you to run javascript at the back-end/ server). Install Node js from here: <https://nodejs.org/en/download/>

\* Again you don’t need to stress yourself with what Node js actually is, consider it something similar to a back-end programming language (Javascript here) like PHP.

Select the LTS version and then your operating system.

Once Node js is installed now run:

npx create-react-app your-app-name

If you encounter an error saying “………………..as it does not contain a package,json file”, run:

npm i create-react-app *(both “i” and “install” can be used alternatively in node commands)*

*npm is a package manager of Node js which allows you to install various node packages create-react-app being one of such.*

*Now run:*

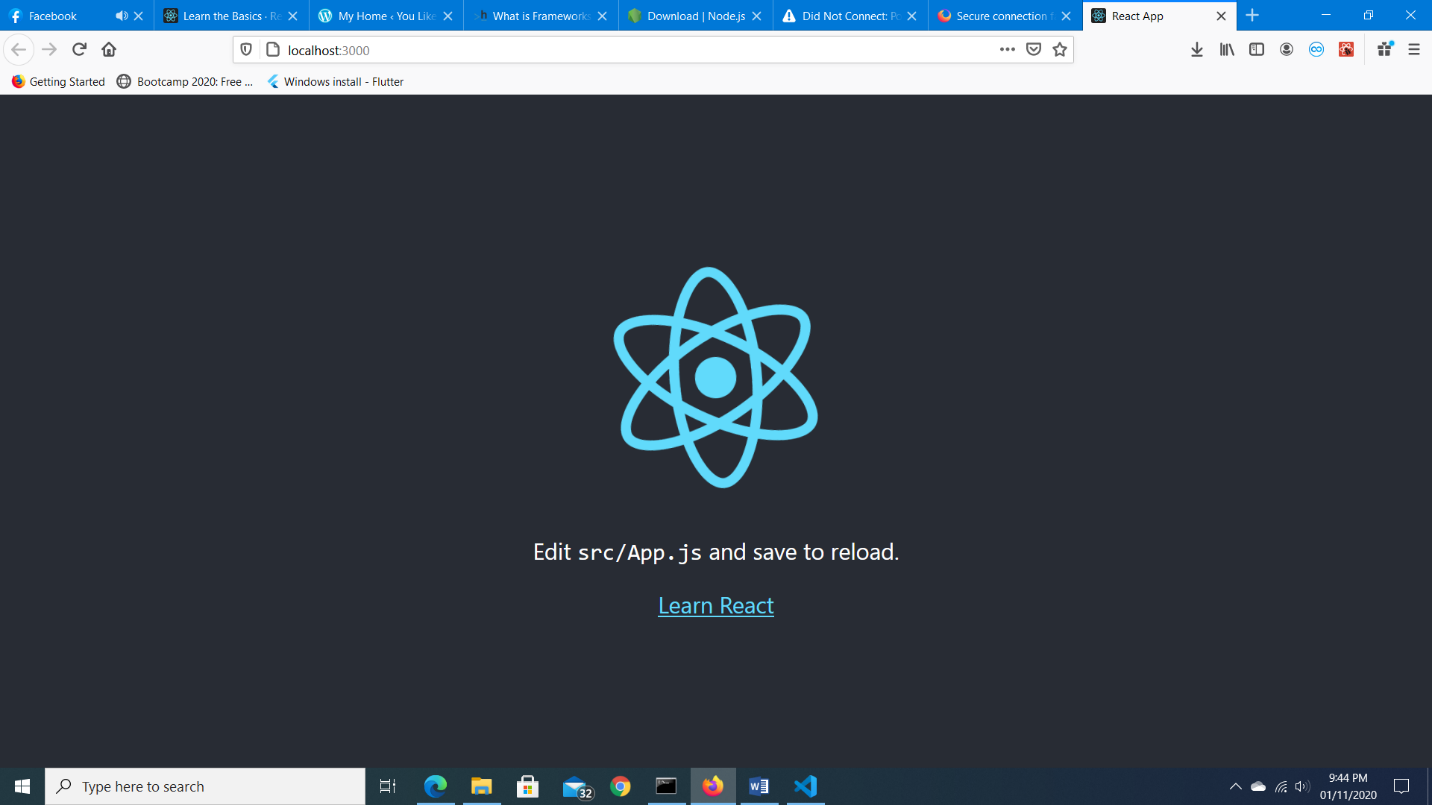
npx create-react-app your-app-name

and your first react app has already been created in the folder named “your-app-name” in this case. You can specify the app name of your choice.

Also run:

npm start

in terminal/ command prompt while in your app’s folder. This would run the development build of your basic react app in the browser at localhost:3000 which would look something like:



Now open that folder in a code editor of your choice.

A few files and folders of our concern for now would be:

The “public” folder containing the index.html

The “src” folder containing the App.js file along with some others.

**The hierarchy**

Let’s now connect some dots.

Open public\index.html in the code editor. Forget the meta and link tags for now and see the body where we have an empty <div> with the id “root”.

Leave that empty and now open: src\index.js which would be having this code:

ReactDOM.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>,

  document.getElementById('root')

);

*Yeah I know there would be something above this too. That code just imports the required modules like React, ReactDOM etc. to compile this code and connects the styling file index.css available in the src folder.*

Anyways so in the code above, a method (coming from ReactDOM which itself is a React module) we see is:

ReactDOM.render(

/\*blah blah blah\*/,

document.getElementById(‘root’)

);

This simply means that render my /\*blah blah blah\*/ stuff on the <div> with the id “root” in that public\index.html file. Try changing this “root” to something else and also change the div id in index.html to the same and the app would still render. (You don’t need to reload the page in your browser, it would reload automatically with the saved changes)

Now what is that:

<React.StrictMode>

    <App />

  </React.StrictMode>,

For the StrictMode part all you need to know now is that it is a React tool which highlights potential problems during development. What we are concerned with here is the <App />.

What is this <App/>? Looks similar to an HTML tag? Well that is because React uses an object oriented language called JSX which is an extension of Javascript. JSX allows us to use HTML tags and HTML like tags in our Javascript files. Again, we do not have to be concerned about the nitty gritty here.

<App/> is a component being exported from App.js (in the same src folder) and imported here in our index.js file through:

import App from './App';

**What are components in React?**

Components are what build a React UI. A React UI is a collection of components out of which some may be parents and some may be children of those parent components which all relate to a single grand parent, the <App /> here.

One component may have several components inside it.

React supports two types of components: class based and functional. We would be focusing on functional components in this tutorial.

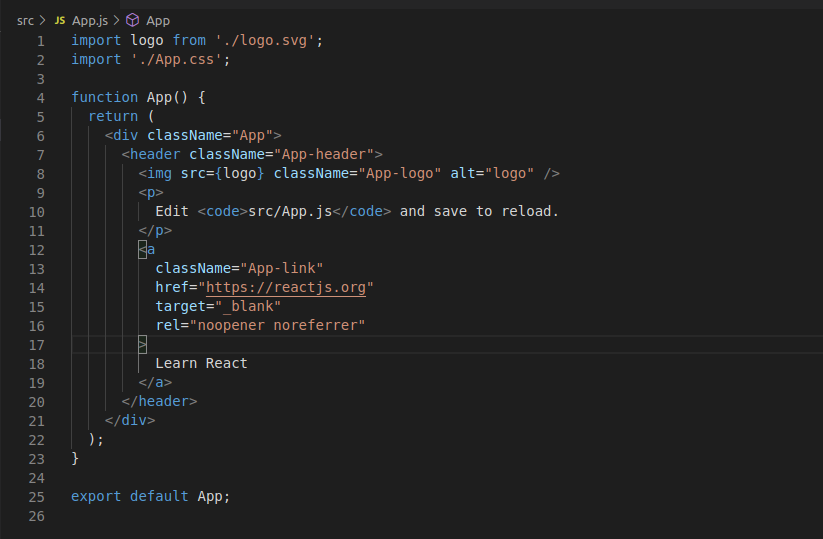
Now we can sum up the code we went through as:

“Render the App component from App.js in StrictMode on the div with the id “root” in our public\index.html file.”

Why all this mess though? Remember, React is Javascript/ JSX not HTML and browsers recognize only HTML for rendering/ displaying stuff. Js needs to connect with HTML somehow to display what it wants. So, it’s actually that index.html file being rendered with stuff coming from js files working in the background through that linkage we just saw.

**Let’s now see components in action**

Open App.js in the src folder and you’ll see some code similar to the one below:



You must have noticed that the return function starting at line 5 contains mainly HTML code with two small differences:

1. Some tags have attributes with the name “className”. This is the same as “class” in HTML, however, as we are actually writing Js here rather than HTML, we use the “className” attribute recognized by Js, instead of “class”.
2. Second difference you might have noticed is the way the “src” attribute in <img> tag refers to logo at line 8. Well that is because, whenever you have to write javascript within HTML you do that within curly braces like {logo} or let’s say {console.log(“something”)}. In React you add images this way:

* import the image as in line 1 (any other name could have been given to the imported image instead of logo). ‘./logo.svg’ means the logo.svg file available in the same folder (this works just the way you have been giving paths in HTML files).

The next import on line 2 is of the App.css file available in the same “src” folder. You may use this file to style your HTML elements inside the return function the way you want. Just remember that instead of class, you have to use className, the rest is same, give them name, id or any other attribute you want and style them with the App.css file. The syntax to be used in this css file would be the same as you use in any other css file.

If you prefer inline styling do it like: style={{backgroundColor:”blue”, display:”flex”, flexGrow:1}}

Noticed the double curly braces? Well the first pair is to tell React that this is some javascript stuff we are putting in and the inner set of curly braces denotes that we have inserted our styling stuff in the shape of a javascript object. We have also used the javascript counterparts of css properties like “backgroundColor” instead of “background-color” and put commas to separate the property value pairs instead of semi-colons because this is a javascript object wherein key value pairs are separated by commas.

Now comes the function (line 4). A while ago I mentioned that we would be focusing on functional components, so here is one for you.

That <App /> we used in the index.js file was basically this function in App.js or in React terms a functional component.

This App function here simply returns some JSX which in simplest terms can be called “HTML or HTML like stuff put into Javascript”. Whatever you have to display/ render on the browser put that in HTML format (with the changes mentioned in (a) and (b) above) inside the return function’s parentheses.

Ok now if you remember we imported this App function in index.js file. For it to be imported there it had to be exported from here, hence the line:

export default App; (at 25)

The “default” keyword ensures that whenever this file is imported anywhere the App function would be exported from it. The function could have been imported with any name and used in index.js and because it was a default export, React would have known that it is the App function we are talking about.

The App function could also have been exported like:

export {App}; //However, this way it would not have been a default export and while importing it had to be specifically mentioned like:

import {App} from ‘./App’;

or we could also have written export default before function App() and could have excluded line 25. You’ll such an example as we move further.

That was the structure of a react app at its most basic level.

**Let’s do our own stuff now**

Let’s build a simple component named Parent which would render a heading <h1> saying “I am the parent here”.

Make a new folder named “components” in your “src” folder. (This is not necessary though, you can make your components anywhere and simply import them where you want by giving the correct path)

Create a new file inside your components folder named Parent.js and write the following code in that:

export default function Parent() {

console.log("parent component rendering")

return (

<h1>I am the parent here</h1>

)

}

The code above has created our functional component named “Parent” and also nominated it as the default export from this file so that it can be used in any other file.

The second line:

console.log("parent component rendering")

is just there to show that it is not only what you have to render on the page that can be written inside the component, you can add whatever javascript you want inside your component. For example, this could have been a function which returned something we could use inside an HTML tag or only some function calculating a value. Will show that later just not to complicate things right now.

Now how to render it on the page?

Remember our App.js is what is being rendered on our page right now. *(If you have lost track, please have a look at the “hierarchy” part above)*

Let’s change our App.js to:

import Parent from './components/Parent';

function App() {

return (

<Parent/>

);

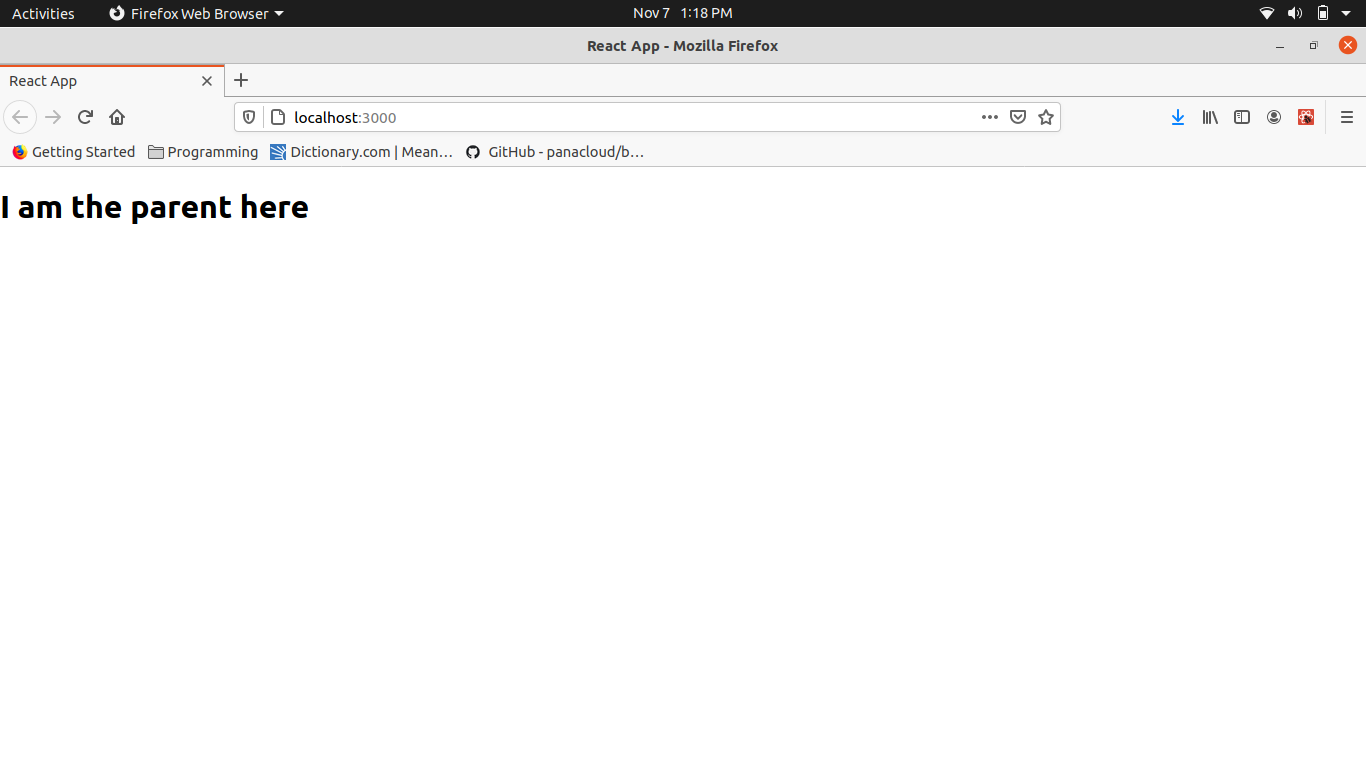
}

export default App;

What we did here is simply:

* Imported our Parent component inside the App component; and
* Called that inside the return parentheses of our App component (The App component in turn is being rendered through index.js on our page, therefore, whatever is returned from it, would be rendered on the page)

The result:

Now let’s add some more javascript to our Parent component. Amend the code in your Parent.js to this:

export default function Parent() {

function givePartyHeading(){

return (<h3>I like to party</h3>)

}

var myRandomNumber;

function giveRandomNumber(){

myRandomNumber = Math.random()\*10

return myRandomNumber

}

giveRandomNumber()

return (

<div>

<h1>I am the parent here</h1>

{givePartyHeading()}

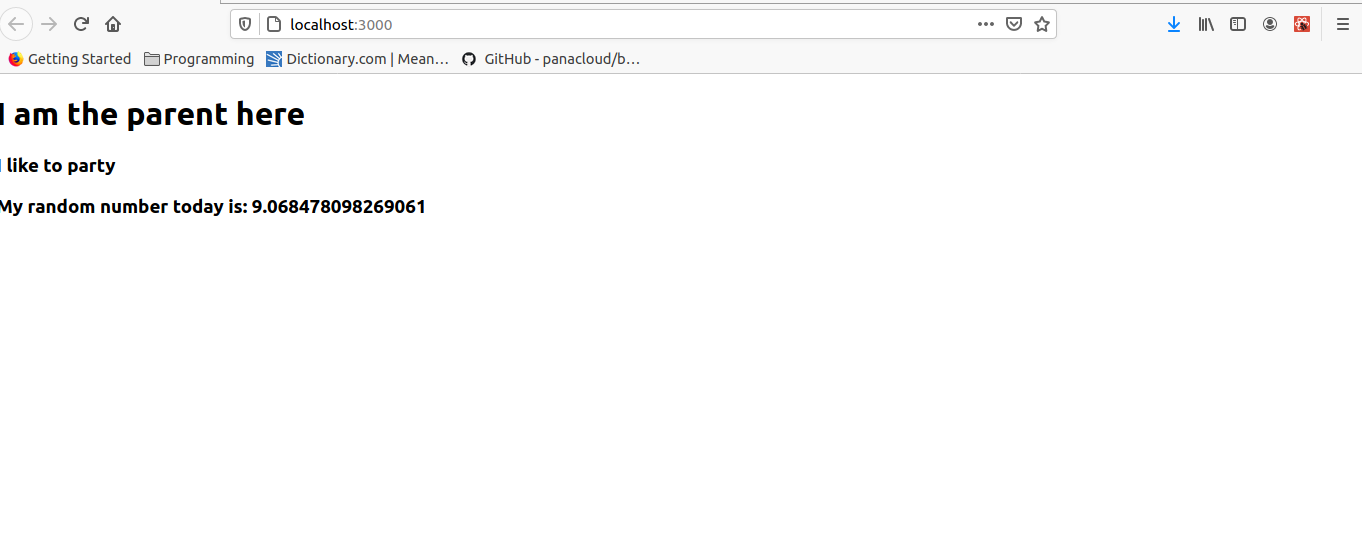
<h3>My random number today is: {myRandomNumber}</h3>

</div>

)

}

**The result:**



Woah! How did our sweet little code turn into this monster? Well, nothing much actually happened, we just did some simple javascript.

I told you we could add whatever javascript we wanted to our components, they are not just there to render HTML content.

So let’s have a closer look at what we did above.

Line 1: Our Parent function is just there as it was with a few additions.

Lines 2 to 4: Next we create a function inside the Parent function (our functional component) which returns some HTML or a JSX element to be more precise i.e. <h3>I like to party</h3>. Then we use this JSX/ HTML element inside our Parent’s return function by calling our giveParentHeading function. See Line 14. Again we used the curly braces to call the function as we used javascript within HTML tags.

Lines 5 to 10: Here we declare a variable first. Then write a function which is to change the value of this variable to a number generated through javascript’s random function from Math module multiplied by 10. The function then would simply return the variable with the new random value. Next we call this function to actually give a random value to our variable “myRandomNumber”. Then finally we use this variable inside our <h3> tag on Line 15 inside the Parent’s return function. Same is the story of curly braces as told above.

Line 12 & 16: The <div> tags. Encompassing our JSX/ HTML elements inside a <div> might be looking unnecessary here but that is not the case. This <div> in fact acts as the parent of all other JSX elements because JSX requires its elements either to be orphan and without sibling i.e. a single element in the component like when we called our Parent component as a single element inside our App component above or if there are more than one components being returned they all should have a single parent (the <div> in our example).

**Let’s now add a Child to our Parent component**

In the same Parent.js file, add the following code:

function FirstChild(){

return (

<div>

<h1>I am the child in the parent's file</h1>

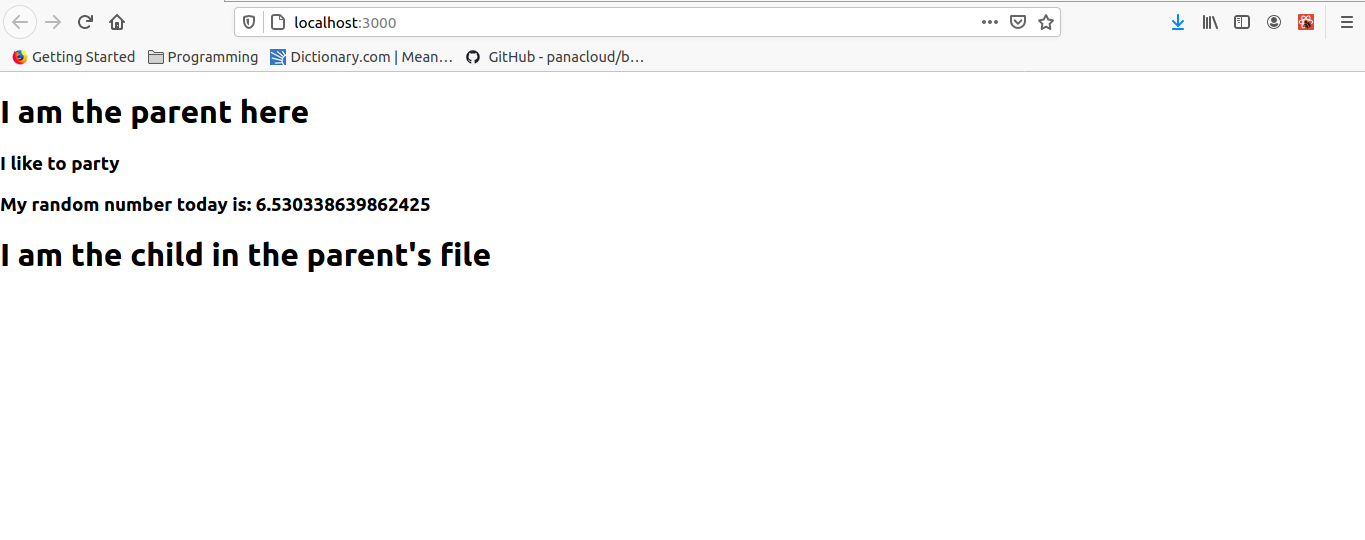
</div>

)

}

This simple function is our new component which would be returned inside our Parent component. To call it inside our Parent component we just add a tag inside its return function like

<FirstChild/> (add this after Line 15 to get the following result):

**Now let’s add another Child from another location**

Inside the src\components folder, create a new file with the name SecondChild.js. Add the following code to it: (or anything you like :-))

export const SecondChild = () => {

return (

<h2>I am the child from another file</h2>

)

}

Oops! Suddenly a change of syntax? Not exactly. It’s just that we used an arrow function (from ES6) instead of the normal one and exported it. Next we will import this inside our Parent.js to render it inside our Parent component. Parent.js would now become:

import { SecondChild } from "./SecondChild";

export default function Parent() {

function givePartyHeading(){

return (<h3>I like to party</h3>)

}

var myRandomNumber;

function giveRandomNumber(){

myRandomNumber = Math.random()\*10

console.log(myRandomNumber)

return myRandomNumber

}

giveRandomNumber()

return (

<div>

<h1>I am the parent here</h1>

{givePartyHeading()}

<h3>My random number today is: {myRandomNumber}</h3>

<FirstChild/>

<SecondChild/>

</div>

)

}

function FirstChild(){

return (

<div>

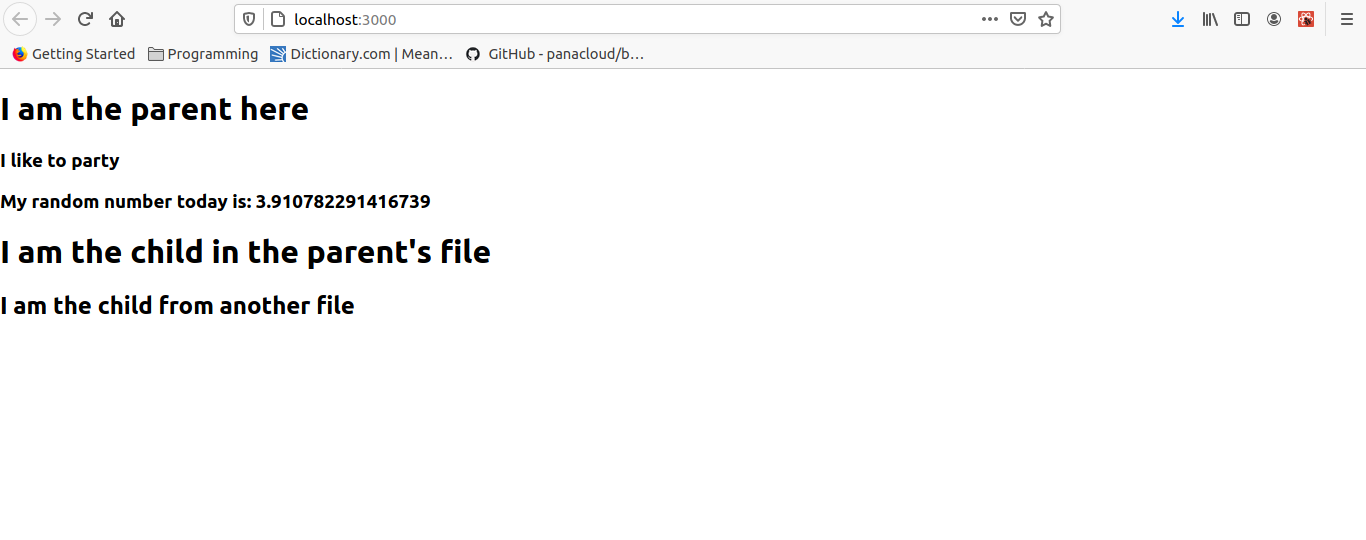
<h1>I am the child in the parent's file</h1>

</div>

)

}

The browser would show:

**Passing data from one component to another (the Props)**

In your Parent.js file, amend the <FirstChild/> and <SecondChild/> calls like this:

<FirstChild value = {23} words = "twenty three"/>

<SecondChild favoriteCharacter = "Wee Willy Winkie"/>

Now we add a parameter named “props” to the FirstChild function and also add some more JSX to its <h1> element. The FirstChild functional component would now look like:

function FirstChild(props){

return (

<div>

<h1>I am the child in the parent's file and got this from the Parent: <span style={{color:"red"}}>{props.value} ({props.words})</span></h1>

</div>

)

}

Similar thing we do with the SecondChild component but here we would name our parameter “char” (no specific reason other than to show that it is not necessary to name your parameter props to received props :-)). Similarly we would also use what we receive in char inside our component’s return function. The SecondChild function would now look similar to this:

export const SecondChild = (char) => {

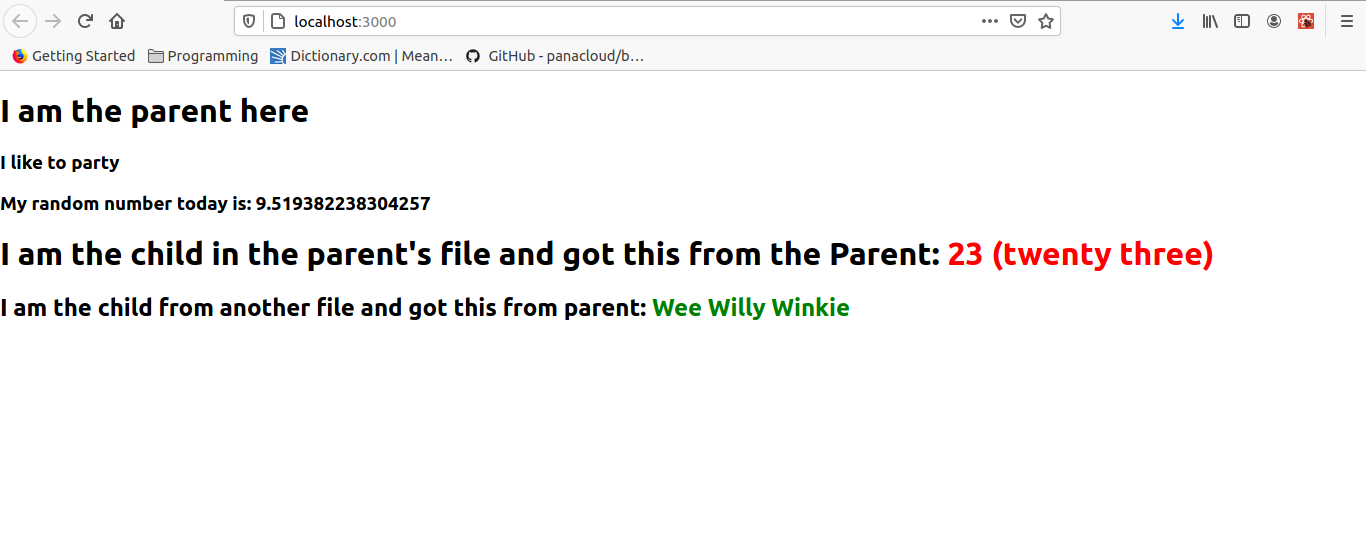
return (

<h2>I am the child from another file and got this from parent: <span style={{color:"green"}}>{char.favoriteCharacter}</span></h2>

)

}

The result in the browser would now be:



So what actually have we done here? We simple passed some props or properties from Parent function/ component to the children.

We passed two things to the FirstChild, a number 23 through the attribute “value” and some text/ string in the attribute “words” which it received in its “props” parameter and created a javascript object like:

props = {value:23, words:”twenty three”}

Hence we had to access these through props.value and props.words.

*If you are concerned with why we entered the number 23 in curly braces and the string/ text without them, just recall how we give attributes their values in HTML; in the form of strings, right? So entering strings without curly braces is fine here but while entering numbers as attributes’ values we have to tell that it is a javascript thing.*

Secondly, we passed only a single string of text to the SecondChild in the attribute “favoriteCharacter” which it received in its “char” parameter and created the object:

char = {favoriteCharacter:”Wee Willy Winkie”}

Again therefore, we used char.favoriteCharacter similar to how we access values of given keys from objects in javascript and hence the curly braces too.

The <span> thing is just for adding styling.

The **state** of a component

Finally! The state. Why is state so important in React?

One of the finest features of React is that it lets us re-render parts of our web app without reloading the page and state helps us do just that.

So, what actually is a state?

A state is some data specified as state which when modified, causes the component to re-render. Remember, each component has its own state and it can only be passed to another component (through props) without that other component adopting that state. That other component would re-render on state change not because the state changed but because the component which called it re-rendered due to its state being changed. Confusing? Don’t worry. Would give a practical example shortly to explain.

Normally, React does not allow global states which can be shared between components but there is a javascript library named “Redux” which let’s you do that. This is just for information, we won’t be discussing Redux here as it would complicate things too much.

So, enough of the theory, let’ see state in action.

Create another component in your components folder and name it ThirdChildWithState.js. Add the following code to it:

const { useState } = require("react");

function ThirdChildWithState(){

const [myState,myStateModifier]=useState(0)

return(

<div>

<h2>I am the Third Child. My state right now is {myState}</h2>

<button onClick={()=>{myStateModifier(myState+1)}}>State Incrementor</button>

</div>

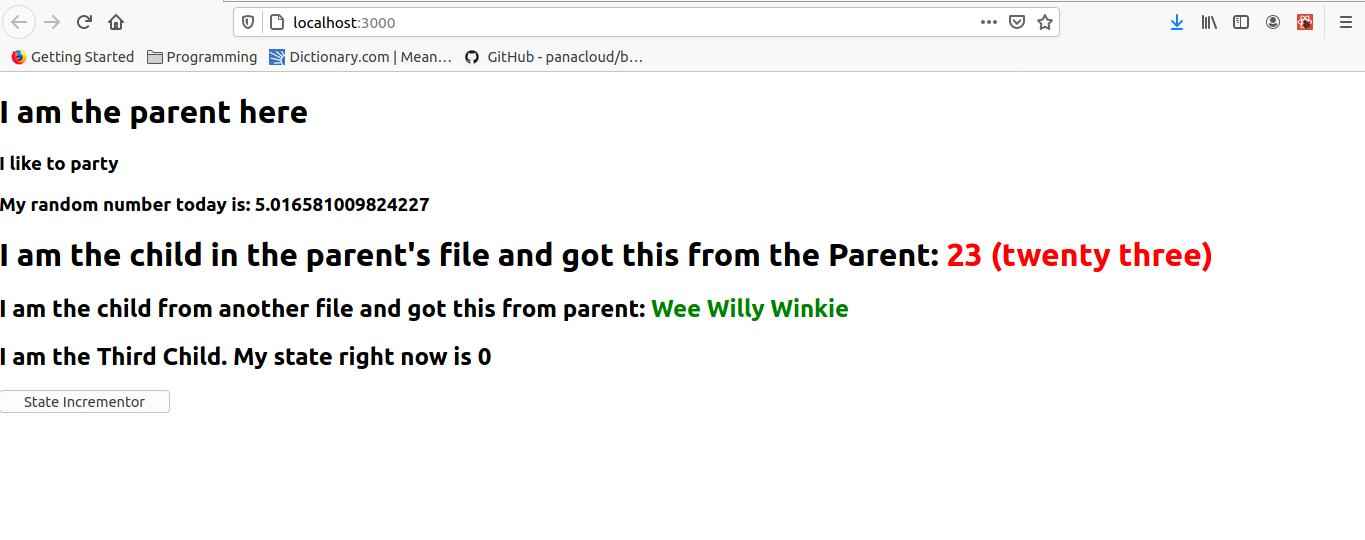
)

}

export default ThirdChildWithState

and call this component inside your Parent component function just like you called the SecondChild after importing it.

The result on the browser would now be:

Hit the “State Incrementor” button and the number 0 would become 1. Hit again and it becomes 2 and so on. But the page would not reload, it would only be the ThirdChildWithState that would be re-rendering.

Let’s analyze the code. (The ThirdChildWithState.js)

We first imported the useState hook from ‘react’ (an in-built library) by:

const { useState } = require("react");

Alternatively the syntax could have been:

import {useState} from 'react';

You can think of Hooks as abstract functions which give you some functionality. React library has some very useful hooks already built-in, however, you can write your own hooks too. Don’t worry about that now, you might never need to do that.

So what this useState hook does is that it returns us an array containing the state at index 0 and a state modifier function at index 1. See what we did here:

const [myState,myStateModifier]=useState(0)

Here the variable myState becomes our component’s state with the initial value 0 we passed in “useState(0)” after the equal sign. MyStateModifier becomes a function which would update our myState variable with the value passed to it whenever called. Check the onClick function of our button:

<button onClick={()=>{myStateModifier(myState+1)}}>State Incrementor</button>

We just assigned an arrow function to the onClick attribute which calls the myStateModifier function by passing it a value which is one plus the current value of our myState variable. Hence the number increments by 1 each time we press the button.

**Passing and using state as props from one component to another**

Now let’s make two more components one that has its own state and passes it to another component which modifies it.

Our two new files in the components folder would be:

FourthComponentThatGivesState.js

import {useState} from 'react';

import FifthChildThatGetsState from './FifthChildThatGetsState';

export default function(){

const [fourthState,fourthStateModifier] = useState(false);

return (

<div>

<h1 style={{color:fourthState==true?"red":"green"}}>I am the fourth child who passes its state to the fifth child</h1>

<FifthChildThatGetsState state={[fourthState,fourthStateModifier]}/>

</div>

)

}

FifthComponentThatGetsState.js

export default function FifthChildThatGetsState(stateAsProp){

return (

<div>

<h1 style={{color:stateAsProp.state[0]==true?"red":"green"}}>I am the fifth child who gets state from fourth child</h1>

<button onClick={()=>stateAsProp.state[1](!stateAsProp.state[0])}>State Changer</button>

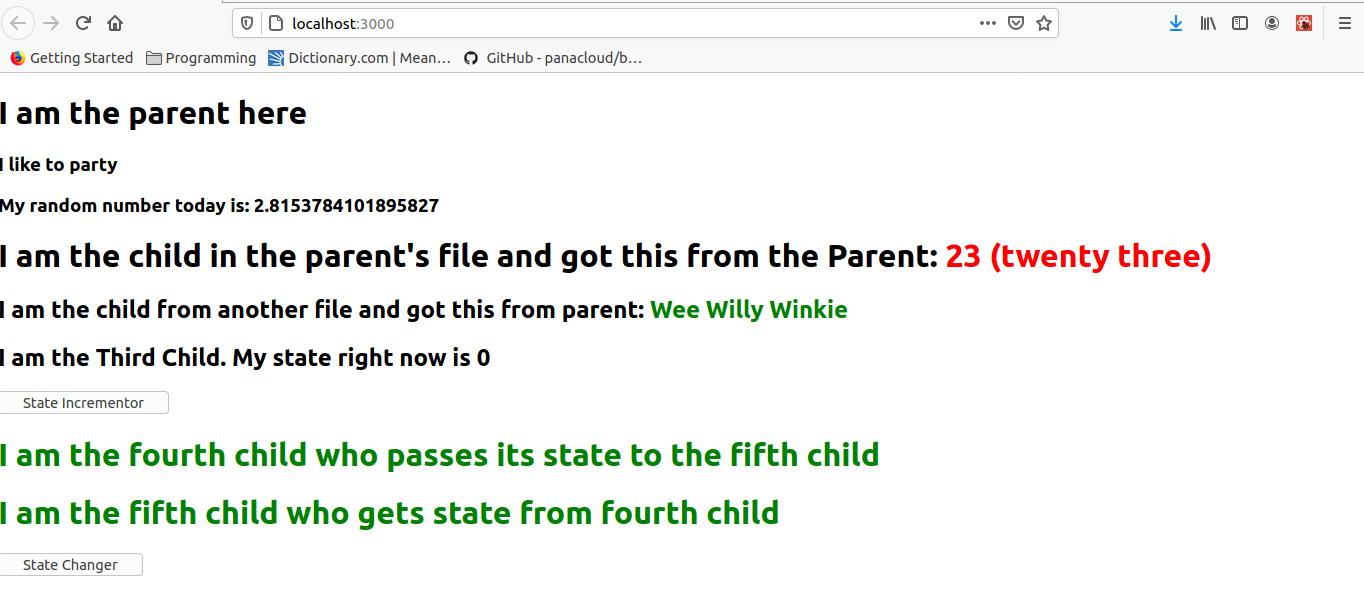
</div>

)

}

Then we would call only the FourthChildThatGivesState inside our Parent component as the FourthChildThatGivesState is already calling FifthChildThatGetsState.

The result on browser:



Hit the “State Changer” button and colors of the last two lines would alternate between red and green.

Let’s look at the code.

FourthChildThatGivesState:

* The component first calls useState to get fourthState as its state variable with the initial value of false and fourthStateModifier as the modifier function of its state.
* It returns an <h1> element whose styling (color property) is based on the state as:

style={{color:fourthState==true?"red":"green"}}

This question mark and colon thing is just the ES6 variant of an if else statement. What it says here is:

if (fourthState==true) {return ”red”} else {return “green”}

* It then passes these two i.e. the state variable and the modifier function bundled in an array as props while calling FifthChildThatGetsState.

FfithChildThatGetsState:

The component gets the state and its modifier function of FourthChildThatGivesState inside its stateAsProp paremeter. Remember, we passed those two inside an array within the attribute “state” from FourthChildThatGivesState.

Also recall that props are saved as objects so here the stateAsProps would become:

stateAsProps = {state: [fourthState,fourthStateModifier]}

Hence inside the FifthChildThatGetsState we access these as:

stateAsProps.state[0] to get the fourthState variable; and

stateAsProps.state[1] to get the fourthStateModifier function.

You can alternatively destructure the array received to variables of your choice, like:

const [stateVariable,stateModifier] = stateAsProp.state

and use the stateVariable and stateModifier instead of stateAsProps.state[0] and stateAsProps.state[1] respectively.

The styling thing in this component is the same as FourthChildThatGivesState.

Look at the arrow function in the button element where we call the fourthStateModifier function as stateAsProps.state[1](!fourthState). This changes the state to the opposite of what it was previously i.e. true if previously false and false if previously true.

The changing of colors of both lines that is one from the FourthChildThatGivesState and the other one from the FifthChildThatGetsState is because:

Whenever we hit the “State Changer” button we made in FifthChildThatGetsState, it updates the state of FourthChildThatGivesState and hence that component re-renders. The color of line coming from FifthChildThatGetsState changes because it is being rendered inside FourthChildThatGivesState which passes it the updated state every time it re-renders while the styling (color) of the line in FifthChildThatGetsState is dependent on the value of the fourthState variable coming from FourthChildThatGivesState.

So that’s about it. Hope I have been able to convey the important concepts in an easily understandable form. For queries feel free to use the contact page.

You can get the full code by cloning/ downloading the repository at:

https://github.com/waqasusmani/react-tutorial

(You would have to run “npm install” inside the project directory once to install the required node modules after cloning/ downloading the repository on your computer)

Thank you.