

React Essentials

Garen Ikezian

What is a Component?

Components in React is a JavaScript function that returns an HTML markup thereby allowing you to create reusable web templates/code. It is one of the key foundations in React and they always start with an uppercase letter to distinguish from vanilla HTML tags.

```
function SomeComponent(){
  return (
    <h1>This is a header</h1>
  )
}

export default function Main(){
  return(
    <SomeComponent/>
  )
}
```

Important stuff to know in React:

- **JSX:** It is an extension of HTML for Javascript. It is a convenient tool to have HTML tags inside a JavaScript file.
- **Babel:** A transpiler to make the new stuff in ECMAScript 2015+ or ES6 (some JavaScript version) to be backwards compatible to ES5. Since Babel can also understand JSX, it is used by React to transpile its components. Ex:

```
// Babel Input: ES2015 arrow function
[1, 2, 3].map(n => n + 1);
```

```
// Babel Output: ES5 equivalent
[1, 2, 3].map(function(n) {
  return n + 1;
});
```

- **Props:** It is short for “properties”. They are static parameters of componenets (they are seen attributes in HTML tags). They are usually made for rendering purposes.

Ex:

```
export default function Main(props) {
  return (
    <section>
      <p>
        We serve the most {props.adjective}{" "}
        food around.
      </p>
    </section>
  );
}

function App() {
  return (
    <div>
      <Main adjective="amazing" />
    </div>
  );
}
```

```
    );  
  }  
}
```

- **State:** It is a dynamic property (unlike regular static properties). It helps a component “remember” information and are used to keep track of something. States are not functions, they are just snapshots of data that changes with every render.

What is a Hook?

Hooks in React are a function and are used to handle states for components. It is a convention for hooks to start with the word “use”.

There are different types of hooks. Some are built-in and some are custom made.

Understanding useState(), useEffect(), useReducer(), and useRef() Hooks

useState()

useState is a hook that adds a state to a component.

Syntax:

```
const [state, setState] = useState("state value");
```

useState returns an array of two values.

1. The value of the state.
2. The set function that lets you change the value of that state.

It’s like a compact getter and setter method in Java or C#.

Usage:

useState() is best suited with user events handler like onClick, onChange, onSelect or any other HTML DOM events.

```
import {useState} from "react";
```

```
function App(){  
  const [name, setName] = useState("John");  
  return(  
    <div>  
      <h1>Hi! my name is {name}</h1>  
      <button onClick={() => setName("Peter")}>Change my name to Peter</button>  
    </div>  
  );  
}
```

Note however that the following is the **WRONG** way of using useState(). setEmotion() will change the value only *after* the rendering is done (or after the return statement):

```
import { useState } from "react";
```

```
function App(){  
  //We create a state like so  
  const [emotion, setEmotion] = useState("Sad");  
  console.log(emotion.value); //emotion is "sad"  
  
  setEmotion("Happy");  
  //Sad does not become "Happy". But this is the wrong way of using setEmotion().  
  console.log(emotion.value); //Still "Sad"  
  
  return (  
    //It will not print anything. The console will show up with errors.  
    <h1>I am {emotion}</h1>  
  )  
}
```

```
);  
}
```

Will lead to:

Too many re-renders. React limits the number of renders to prevent an infinite loop.

Since `setEmotion()` is called for every rendering on loop, the browser will then complain that it is “re-rendering” too much. From the browser’s point of view, this is what it looks like:

```
function render(){  
  render();  
}  
  
render();
```

Because of this, we need a user event handler to prevent an infinite recursion. `setEmotion()` is supposed to be called only once every time the button is clicked.

useReducer()

`useReducer()` is like `useState()`. But unlike `useState()`, `useReducer()` accepts static logic (represented as a function) as its parameter.

```
const [state, function] = useReducer(reducer, initialState)
```

Unlike having state logics being spread out throughout the code with `useState()`, `useReducer()` helps organize different state logic into its respective methods.

Usage:

```
function reducer(state, action) {  
  return { condition: !state.condition};  
}  
  
function App(){  
  const [checked, toggleChecked] = useReducer(reducer, { condition: false });  
  
  return (  
    <div>  
      <input  
        type="checkbox"  
        checked={checked.condition}  
        onChange={toggleChecked}  
      />  
      <label>{checked.condition? "checked" : "not checked"}</label>  
    </div>  
  );  
}
```

Or you can use an arrow function to make the code A LOT more legible.

```
function App(){  
  const [checked, toggleChecked] = useReducer((checked) => !checked, false );  
  return (  
    <div>  
      <input  
        type="checkbox"  
        checked={checked}  
        onChange={toggleChecked}  
      />  
      <label>{checked? "checked" : "not checked"}</label>  
    </div>  
  );  
}
```

useEffect()

`useEffect()` is a hook that accepts a function and an optional list.

With `useState()`, you (re-)initialize a component with a state. With `useEffect()`, you can perform side effects like directly fetching or updating the state of a component. It is made to output unpredictable results for the user.

`useEffect()` is made to address the common misuse of `useState()`. However, its intended use is mainly for dealing data with third-party tools like backend servers, browser APIs, and timing functions like `setTimeout()` and `setInterval()`.

In short, if you want an expected outcome, use `useState()`. If you want the webpage to interact with the outside world or expect unexpected results, use `useEffect()`.

If your code does not involve (a)synchronization, you do not need `useEffect()`. More here.

There are three possible usage scenarios:

1. Without a dependency (2nd param). **BAD USAGE**

```
useEffect( () => {  
  //1st param: code that runs after *every* render  
});
```

2. With an empty list

```
useEffect( () => {  
  //1st param: code that runs only on the first render  
}, []);
```

3. With a list

```
useEffect( () => {  
  //1st param: code that runs on the first render then after any dependency value change  
},  
  //2nd param: A dependency array of states specified. If its values change, the code in the 1st param above w  
[...,...,...]);
```

Usage:

To demonstrate how `useEffect()` looks like without a dependency, We use a `setTimeout()` method:

```
useEffect(() => {  
  setTimeout(() => {  
    setCount((count) => count + 1);  
  }, 1000);  
});  
  
return <h1>I've rendered {count} times!</h1>;
```

The rendering does not stop as the dependency does not exist. So we have to add an empty array as our second parameter.

```
useEffect(() => {  
  setTimeout(() => {  
    setCount((count) => count + 1);  
  }, 1000);  
}, []);  
  
return <h1>I've rendered {count} times!</h1>;
```

Now it only renders once.

useRef()

All of the hooks mentioned above will require re-rendering when their values are changed. `useRef()` however, is the exception.

“ref” is short for reference. It lets you “reference”/remember a value without the need of re-rendering.

```
const ref = useRef(initialValue)
```

Note however that states and refs are two different things. `useRef()` does not rely on states as states trigger re-rendering when they are changed. A ref is just a plain JavaScript that can store a value that is used for later use.

If we pass 0 to `useRef()`,

```
const ref = useRef(0);
```

`useRef()` returns an object like so (the browser will “see” this):

```
{
  current: 0 //the value passed to useRef
}
```

“curent” is just an attribute of ref (`ref.current`).

Usage

Here, we hold the number of times the user clicked on a button. Unlike `useState()`, it does not trigger a re-render when the value is changed.

```
import { useRef } from 'react';

export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked '+ref.current+' times!');
  }

  return (
    <div>
      <h1>You clicked me {ref.current} time(s)!</h1>
      <button onClick={handleClick}>
        Click me!
      </button>
    </div>
  );
}
```

Notice how the value of `ref.current` in the return statement is not updated. Refs do not re-render its components when they are re-rendered. They only serve as a storage space for later use.

This [table](#) from the manual is excellent to distinguish between refs and states apart.

Fetching Data with Hooks

There is a link to output a user’s data on Github. Let’s the take the following example:

```
https://api.github.com/users/Garenium
```

This shows a user’s Github data respresented as a JSON object. In order to fetch this with React, we use `useEffect()` like so:

```
import './App.css';
import { useState, useEffect } from 'react';
import { useReducer } from 'react';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(
      `https://api.github.com/users/Garenium`
    ).then((response) => response.json())
    .then(setData);
  });
}
```

```

}, []);
if(data) return <pre>{JSON.stringify(data,null,2)}</pre>
return (
  <h1>Data</h1>
);
}

```

export default App;

The problem with this code is that it cannot handle three different states:

- When the page is loading
- When the page is finished loading
- When there is a fetch problem

Notice that when the webpage was loading, it was showing “Data” in a blink of an eye instead of the actual JSON object. This is because it is the asynchronous nature of the `useEffect` hook. If we were to encounter a Github username that didn't exist, we would see this:

```

{
  "message": "Not Found",
  "documentation_url": "https://docs.github.com/rest/users/users#get-a-user"
}

```

Figure 1: Output when a Github user doesn't exist

It is recommended to create three different states like so:

```

import "./App.css";
import { useState, useEffect } from "react";

//Create a separate component for extract Github user data after useEffect is
//done
function GithubUser({ name, location, avatar }) {
  return (
    <div>
      <h1>{name}</h1>
      <p>{location}</p>
      <img src={avatar} height={150} alt={name} />
    </div>
  );
}

function App() {
  //The three states:
  const [data, setData] = useState(null); //Data to load fetch
  const [error, setError] = useState(null); //setError when fetching fails
  const [loading, setLoading] = useState(false); //setLoading before data is set
  useEffect(() => {
    setLoading(true);
    fetch(
      `https://api.github.com/users/moonhighway`
    )
      .then((response) => response.json())
      .then(setData)
      .then(() => setLoad(false))
      .catch(setError);
  }, []);
  if(loading) return <h1>Loading...</h1> //Show this when the state is loading
  if(error) return <pre>{JSON.stringify(error)}</pre> //When fetching fails. stringify error

```

```
if(!data) return null; //return null when there's no data

return (
  <GithubUser
    name={data.name}
    location={data.location}
    avatar={data.avatar_url}
  />
);
}

export default App;
```