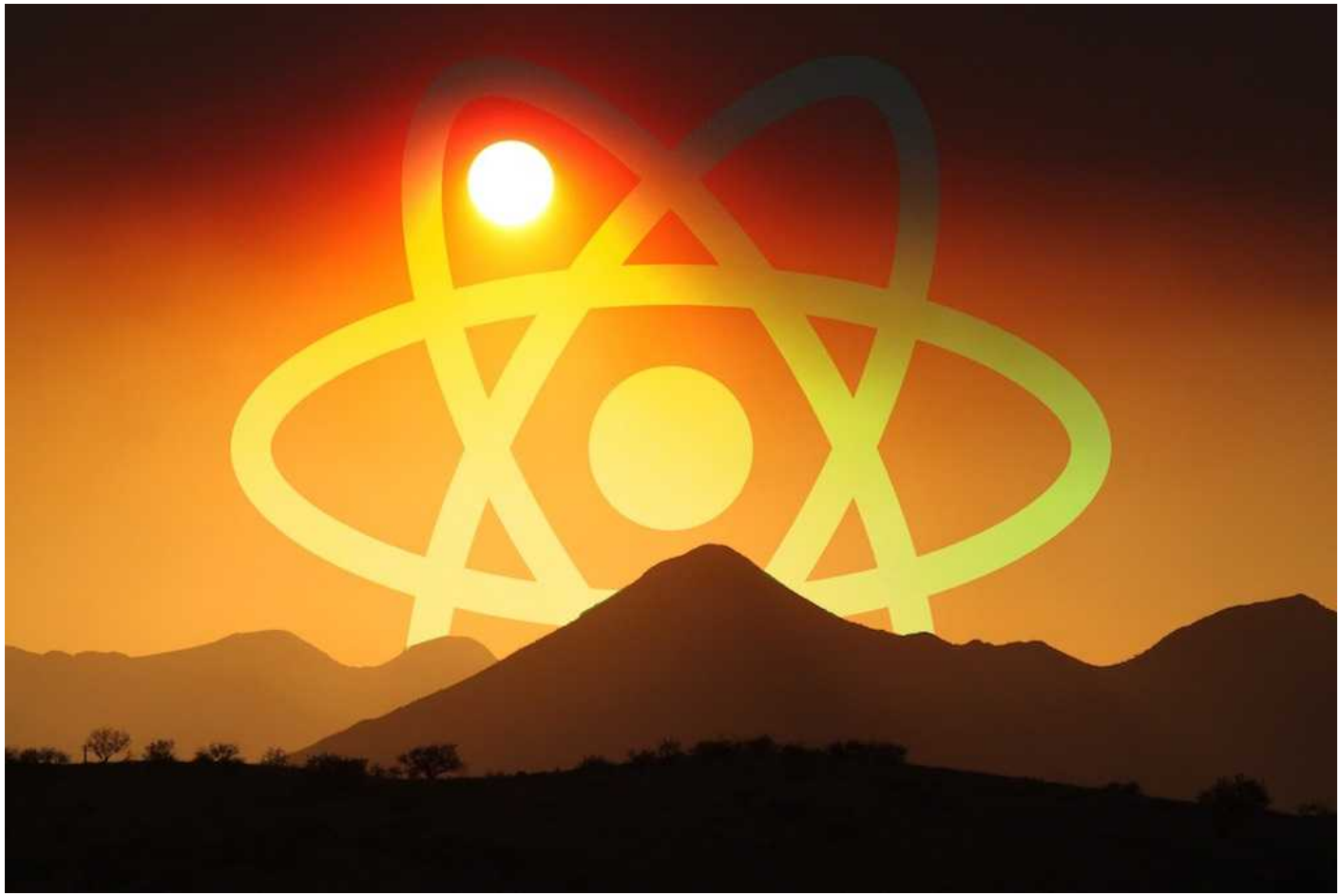
React Function Components

MARCH 18, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook



React Function Components -- also known as React Functional Components -- are the status

Types, but with the introduction of React Hooks it's possible to write your entire application with just functions as React components.

This in-depth guide shows you everything about React Function Components -- which are basically **just JavaScript Functions being React Components** which return JSX (React's Syntax) -- so that after you have read this tutorial you should be well prepared to implement modern React applications with them.

Note: There are several synonyms for this kind of component in React. You may have seen different variations such as "React Function only Component" or "React Component as Function".

TABLE OF CONTENTS

- React Function Component: ref
- React Function Component: PropTypes
- React Function Component: TypeScript
- React Function Component vs Class Component

REACT FUNCTION COMPONENT EXAMPLE

Let's start with a simple example of a Functional Component in React defined as App which returns JSX:

```
import React from 'react';

function App() {
  const greeting = 'Hello Function Component!';

  return <h1>{greeting}</h1>;
}

export default App;
```

That's already the essential React Function Component Syntax. The definition of the component happens with just a JavaScript Function which has to return JSX -- React's syntax for defining a mix of HTML and JavaScript whereas the JavaScript is used with curly braces within the HTML. In our case, we render a variable called *greeting*, which is defined in the component's function body, and is returned as HTML headline in JSX.

Note: If you are familiar with React Class Components, you may have noticed that a Functional

Now, if you want to render a React Component inside a Function Component, you define another component and render it as HTML element with JSX within the other component's body:

```
import React from 'react';

function App() {
  return <Headline />;
}

function Headline() {
  const greeting = 'Hello Function Component!';

  return <h1>{greeting}</h1>;
}

export default App;
```

Basically you have a function as Child Component now. Defining React Components and rendering them within each other makes Composition in React possible. You can decide where to render a component and how to render it.

REACT FUNCTION COMPONENT: PROPS

Let's learn about a React Function Component with props. In React, props are used to pass information from component to component. If you don't know about props in React, cross-read the linked article. Essentially props in React are always passed down the component tree:

```
import React from 'react';
```

```
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
}

function Headline(props) {
  return <h1>{props.value}</h1>;
}

export default App;
```

Props are the React Function Component's parameters. Whereas the component can stay generic, we decide from the outside what it should render (or how it should behave). When rendering a component (e.g. Headline in App component), you can pass props as HTML attributes to the component. Then in the Function Component the props object is available as argument in the function signature.

Since props are always coming as object, and most often you need to extract the information from the props anyway, JavaScript object destructuring comes in handy. You can directly use it in the function signature for the props object:

```
import React from 'react';

function App() {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
}

function Headline({ value }) {
  return <h1>{value}</h1>;
}
```

Note: If you forget the JavaScript destructuring and just access props from the component's function signature like `function Headline(value1, value2) { ... }`, you may see a "props undefined"-messages. It doesn't work this way, because props are always accessible as first argument of the function and can be destructured from there: `function Headline({ value1, value2 }) { ... }`.

If you want to learn more tricks and tips about React props, again check out the linked article from the beginning of this section. There you will learn about cases where you don't want to destructure your props and simply pass them to the next child component with the ...syntax known as spread operator.

REACT ARROW FUNCTION COMPONENT

With the introduction of JavaScript ES6, new coding concepts were introduced to JavaScript and therefore to React. For instance, a JavaScript function can be expressed as lambda (arrow function). That's why a Function Component is sometimes called Arrow Function Components (or maybe also Lambda Function Component). Let's see our refactored React Component with an Arrow Function:

```
import React from 'react';

const App = () => {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
};
```

```
};

export default App;
```

Both React Arrow Function Components use a function block body now. However, the second component can be made more lightweight with a concise body for the function, because it only returns the output of the component without doing something else in between. When leaving away the curly braces, the explicit return becomes an implicit return and can be left out as well:

```
import React from 'react';

const App = () => {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
};

const Headline = ({ value }) =>
  <h1>{value}</h1>;

export default App;
```

When using arrow functions for React components, nothing changes for the props. They are still accessible as arguments as before. It's a React Function Component with ES6 Functions expressed as arrows instead of ES5 Functions which are the more default way of expressing functions in JS.

Note: If you run into a "React Component Arrow Function Unexpected Token" error, make sure that JavaScript ES6 is available for your React application. Normally when using create-react-app this should be given, otherwise, if you set up the project yourself, Babel is enabling ES6 and

REACT STATELESS FUNCTION COMPONENT

Every component we have seen so far can be called Stateless Function Component. They just receive an input as props and return an output as JSX: `(props) => JSX`. The input, only if available in form of props, shapes the rendered output. These kind of components don't manage state and don't have any side-effects (e.g. accessing the browser's local storage). People call them Functional Stateless Components, because they are stateless and expressed by a function. However, React Hooks made it possible to have state in Function Components.

REACT FUNCTION COMPONENT: STATE

React Hooks made it possible to use state (and side-effects) in Function Components. Finally we can create a React Function Component with state! Let's say we moved all logic to our other Function Component and don't pass any props to it:

```
import React from 'react';

const App = () => {
  return <Headline />;
};

const Headline = () => {
  const greeting = 'Hello Function Component!';

  return <h1>{greeting}</h1>;
};
```

So far, a user of this application has no way of interacting with the application and thus no way of changing the greeting variable. The application is static and not interactive at all. State is what makes React components interactive; and exciting as well. A React Hook helps us to accomplish it:

```
import React, { useState } from 'react';

const App = () => {
  return <Headline />;
};

const Headline = () => {
  const [greeting, setGreeting] = useState(
    'Hello Function Component!'
  );

  return <h1>{greeting}</h1>;
};

export default App;
```

The useState hook takes an initial state as parameter and returns an array which holds the current state as first item and a function to change the state as second item. We are using JavaScript array destructuring to access both items with a shorthand expression. In addition, the destructuring let's us name the variables ourselves.

Let's add an input field to change the state with the `setGreeting()` function:

```
import React, { useState } from 'react';

const App = () => {
```

```
const Headline = () => {
  const [greeting, setGreeting] = useState(
    'Hello Function Component!'
  );

  return (
    <div>
      <h1>{greeting}</h1>

      <input
        type="text"
        value={greeting}
        onChange={event => setGreeting(event.target.value)}
      />
    </div>
  );
};

export default App;
```

By providing an event handler to the input field, we are able to do something with a callback function when the input field changes its value. As argument of the callback function we receive a synthetic React event which holds the current value of the input field. This value is ultimately used to set the new **state** for the Function Component with an inline arrow function. We will see later how to extract this function from there.

*Note: The input field receives the value of the component **state** too, because you want to control the **state** (value) of the input field and don't let the native HTML element's internal **state** take over. Doing it this way, the component has become a controlled component.*

As you have seen, React Hooks enable us to use **state** in React (Arrow) Function Components. Whereas you would have used a setState method to write state in a Class Component, you can

Note: If you want to use React's Context in Function Components, check out React's Context Hook called useContext for reading from React's Context in a component.

REACT FUNCTION COMPONENT: EVENT HANDLER

In the previous example you have used an *onChange* event handler for the input field. That's appropriate, because you want to be notified every time the internal value of the input field has changed. In the case of other HTML form elements, you have several other React event handlers at your disposal such as onClick, onMouseDown, and onBlur.

Note: The onChange event handler is only one of the handlers for HTML form elements. For instance, a button would offer an onClick event handler to react on click events.

So far, we have used an arrow function to inline the event handler for out input field. What about extracting it as standalone function inside the component? It would become a named function then:

```
import React, { useState } from 'react';

const App = () => {
  return <Headline />;
};

const Headline = () => {
  const [greeting, setGreeting] = useState(
    'Hello Function Component!'
  );

  const handleChange = event => setGreeting(event.target.value);
```

```
      <div>
        <h1>{greeting}</h1>

        <input type="text" value={greeting} onChange={handleChange} />
      </div>
    );
};

export default App;
```

We have used an arrow function to define the function within the component. If you have used class methods in React Class Components before, this way of defining functions inside a React Function Component is the equivalent. You could call it the "React Function Component Methods"-equivalent to class components. You can create or add as many functions inside the Functional Component as you want to act as explicit event handlers or to encapsulate other business logic.

REACT FUNCTION COMPONENT: CALLBACK FUNCTION

Everything happens in our Child Function Component. There are no props passed to it, even though you have seen before how a string variable for the greeting can be passed from the Parent Component to the Child Component. Is it possible to pass a function to a component as prop as well? Somehow it must be possible to call a component function from the outside! Let's see how this works:

```
import React, { useState } from 'react';
```

```
  const [greeting, setGreeting] = useState(
    'Hello Function Component!'
  );

  const handleChange = event => setGreeting(event.target.value);

  return (
    <Headline headline={greeting} onChangeHeadline={handleChange} />
  );
};

const Headline = ({ headline, onChangeHeadline }) => (
  <div>
    <h1>{headline}</h1>

    <input type="text" value={headline} onChange={onChangeHeadline} />
  </div>
);

export default App;
```

That's all to it. You can pass a function to a Child Component and handle what happens up in the Parent Component. You could also execute something in between in the Child Component (Headline component) for the `onChangeHeadline` function -- like trimming the value -- to add extra functionality inside the Child Component. That's how you would be able to call a Child Component's function from a Parent Component.

Let's take this example one step further by introducing a Sibling Component for the Headline component. It could be an abstract Input component:

```
import React, { useState } from 'react';
```

```
  );

  const handleChange = event => setGreeting(event.target.value);

  return (
    <div>
      <Headline headline={greeting} />

      <Input value={greeting} onChangeInput={handleChange}>
        Set Greeting:
      </Input>
    </div>
  );
};

const Headline = ({ headline }) => <h1>{headline}</h1>;

const Input = ({ value, onChangeInput, children }) => (
  <label>
    {children}
    <input type="text" value={value} onChange={onChangeInput} />
  </label>
);

export default App;
```

I find this is a perfect yet minimal example to illustrate how to pass functions between components as props; and more importantly how to share a function between components. You have one Parent Component which manages the logic and two Child Components -- which are siblings -- that receive props. These props can always include a callback function to call a function in another component. Basically that's how it's possible to call a function in different components in React.

Override Component Function with React

It's shouldn't happen often, but I have heard people asking me this question. How would you override a component's function? You need to take the same approach as for overriding any other passed prop to a component by giving it a default value:

```
import React from 'react';

const App = () => {
  const sayHello = () => console.log('Hello');

  return <Button handleClick={sayHello} />;
};

const Button = ({ handleClick }) => {
  const sayDefault = () => console.log('Default');

  const onClick = handleClick || sayDefault;

  return (
    <button type="button" onClick={onClick}>
      Button
    </button>
  );
};

export default App;
```

You can assign the default value in the function signature for the destructuring as well:

```
import React from 'react';

const App = () => {
  const sayHello = () => console.log('Hello');
```

```
const Button = ({ handleClick = () => console.log('Default') }) => (
  <button type="button" onClick={handleClick}>
    Button
  </button>
);

export default App;
```

You can also give a React Function Component default props -- which is another alternative:



```
import React from 'react';

const App = () => {
  const sayHello = () => console.log('Hello');

  return <Button handleClick={sayHello} />;
};

const Button = ({ handleClick }) => (
  <button type="button" onClick={handleClick}>
    Button
  </button>
);

Button.defaultProps = {
  handleClick: () => console.log('Default'),
};

export default App;
```

All of these approaches can be used to define default props (in this case a default function), to be able to override it later from the outside by passing an explicit prop (e.g. function) to the

Async Function in Component with React

Another special case may be an async function in a React component. But there is nothing special about it, because it doesn't matter if the function is asynchronously executed or not:

```
import React from 'react';

const App = () => {
  const sayHello = () =>
    setTimeout(() => console.log('Hello'), 1000);

  return <Button handleClick={sayHello} />;
};

const Button = ({ handleClick }) => (
  <button type="button" onClick={handleClick}>
    Button
  </button>
);

export default App;
```

The function executes delayed without any further instructions from your side within the component. The component will also rerender asynchronously in case props or state have changed. Take the following code as example to see how we set state with a artificial delay by using `setTimeout`:

```
import React, { useState } from 'react';

const App = () => {
  const [count, setCount] = useState(0);
```

```
      () => setCount(currentCount => currentCount + 1),
      1000
    );

  const handleDecrement = () =>
    setTimeout(
      () => setCount(currentCount => currentCount - 1),
      1000
    );

  return (
    <div>
      <h1>{count}</h1>

      <Button handleClick={handleIncrement}>Increment</Button>
      <Button handleClick={handleDecrement}>Decrement</Button>
    </div>
  );
};

const Button = ({ handleClick, children }) => (
  <button type="button" onClick={handleClick}>
    {children}
  </button>
);

export default App;
```

Also note that we are using a callback function within the `setCount` state function to access the current state. Since setter functions from `useState` are executed asynchronously by nature, you want to make sure to perform your state change on the current state and not on any stale state.

Experiment: If you wouldn't use the callback function within the State Hook, but rather act upon the count variable directly (e.g. `setCount(count + 1)`), you wouldn't be able to increase the

Read more about how to fetch data with Function Components with React Hooks.

REACT FUNCTION COMPONENT: LIFECYCLE

If you have used React Class Components before, you may be used to lifecycle methods such as componentDidMount, componentWillUnmount and shouldComponentUpdate. You don't have these in Function Components, so let's see how you can implement them instead.

First of all, you have no constructor in a Function Component. Usually the constructor would have been used in a React Class Component to allocate initial state. As you have seen, you don't need it in a Function Component, because you allocate initial state with the useState hook and set up functions within the Function Component for further business logic:

```
import React, { useState } from 'react';

const App = () => {
  const [count, setCount] = useState(0);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  return (
    <div>
      <h1>{count}</h1>

      <button type="button" onClick={handleIncrement}>
        Increment
```

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

export default App;
```

React Functional Component: Mount

Second, there is the mounting lifecycle for React components when they are rendered for the first time. If you want to execute something when a React Function Component **did mount**, you can use the useEffect hook:

```
import React, { useState, useEffect } from 'react';

const App = () => {
  const [count, setCount] = useState(0);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  useEffect(() => setCount(currentCount => currentCount + 1), []);

  return (
    <div>
      <h1>{count}</h1>

      <button type="button" onClick={handleIncrement}>
        Increment
```

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

export default App;
```

If you try out this example, you will see the count 0 and 1 shortly displayed after each other. The first render of the component shows the count of 0 from the initial state -- whereas after the component did mount actually, the Effect Hook will run to set a new count state of 1.

It's important to note the empty array as second argument for the Effect Hook which makes sure to trigger the effect only on component load (mount) and component unload (unmount).

Experiment: If you would leave the second argument of the Effect Hook empty, you would run into an infinite loop of increasing the count by 1, because the Effect Hook always runs after state has changed. Since the Effect Hook triggers another state change, it will run again and again to increase the count.

React Functional Component: Update

Every time incoming props or state of the component change, the component triggers a rerender to display the latest status quo which is often derived from the props and state. A render executes everything within the Function Component's body.

Note: In case a Function Component is not updating properly in your application, it's always a good first debugging attempt to console log state and props of the component. If both don't change, there is no new render executed, and hence you don't see a console log of the output in the first place.

```
const App = () => {
  console.log('Does it render?');

  const [count, setCount] = useState(0);

  console.log(`My count is ${count}!`);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  return (
    <div>
      <h1>{count}</h1>

      <button type="button" onClick={handleIncrement}>
        Increment
      </button>
      <button type="button" onClick={handleDecrement}>
        Decrement
      </button>
    </div>
  );
};

export default App;
```

If you want to act upon a rerender, you can use the Effect Hook again to do something after the component did update:

```
import React, { useState, useEffect } from 'react';
```

```
  const [count, setCount] = useState(initialCount);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  useEffect(() => localStorage.setItem('storageCount', count));

  return (
    <div>
      <h1>{count}</h1>

      <button type="button" onClick={handleIncrement}>
        Increment
      </button>
      <button type="button" onClick={handleDecrement}>
        Decrement
      </button>
    </div>
  );
};

export default App;
```

Now every time the Function Component rerenders, the count is stored into the browser's local storage. Every time you fresh the browser page, the count from the browser's local storage, in case there is a count in the storage, is set as initial state.

You can also specify when the Effect Hook should run depending on the variables you pass into the array as second argument. Then every time one of the variables change, the Effect Hook runs. In this case it makes sense to store the count only if the count has changed:

```
const App = () => {
  const initialCount = +localStorage.getItem('storageCount') || 0;
  const [count, setCount] = useState(initialCount);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  useEffect(() => localStorage.setItem('storageCount', count), [
    count,
  ]);

  return (
    <div>
      <h1>{count}</h1>

      <button type="button" onClick={handleIncrement}>
        Increment
      </button>
      <button type="button" onClick={handleDecrement}>
        Decrement
      </button>
    </div>
  );
};

export default App;
```

By using the second argument of the Effect Hook with care, you can decide whether it runs:

- every time (no argument)
- only on mount and unmount ([] argument)

Note: A React Function Component force update can be done by using this neat trick. However, you should be careful when applying this pattern, because maybe you can solve the problem a different way.

PURE REACT FUNCTION COMPONENT

React Class Components offered the possibility to decide whether a component has to rerender or not. It was achieved by using the PureComponent or shouldComponentUpdate to avoid performance bottlenecks in React by preventing rerenders. Let's take the following extended example:

```
import React, { useState } from 'react';

const App = () => {
  const [greeting, setGreeting] = useState('Hello React!');
  const [count, setCount] = useState(0);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);

  const handleChange = event => setGreeting(event.target.value);

  return (
    <div>
      <input type="text" onChange={handleChange} />

      <Count count={count} />
```

```
        Increment
      </button>
      <button type="button" onClick={handleDecrement}>
        Decrement
      </button>
    </div>
  );
};

const Count = ({ count }) => {
  console.log('Does it (re)render?');

  return <h1>{count}</h1>;
};

export default App;
```

In this case, every time you type something in the input field, the App component updates its state, rerenders, and rerenders the Count component as well. React memo -- which is one of React's top level APIs -- can be used for React Function Components to prevent a rerender when the incoming props of this component haven't changed:

```
import React, { useState, memo } from 'react';

const App = () => {
  const [greeting, setGreeting] = useState('Hello React!');
  const [count, setCount] = useState(0);

  const handleIncrement = () =>
    setCount(currentCount => currentCount + 1);

  const handleDecrement = () =>
    setCount(currentCount => currentCount - 1);
```

```
  return (
    <div>
      <input type="text" onChange={handleChange} />

      <Count count={count} />

      <button type="button" onClick={handleIncrement}>
        Increment
      </button>
      <button type="button" onClick={handleDecrement}>
        Decrement
      </button>
    </div>
  );
};

const Count = memo(({ count }) => {
  console.log('Does it (re)render?');

  return <h1>{count}</h1>;
});

export default App;
```

Now, the Count component doesn't update anymore when the user types something into the input field. Only the App component rerenders. This performance optimization shouldn't be used as default though. I would recommend to check it out when you run into issues when the rerendering of components takes too long (e.g. rendering and updating a large list of items in a Table component).

REACT FUNCTION COMPONENT: EXPORT AND

Eventually you will separate components into their own files. Since React Components are functions (or classes), you can use the standard import and export statements provided by JavaScript. For instance, you can define and export a component in one file:

```
// src/components/Headline.js

import React from 'react';

const Headline = (props) => {
  return <h1>{props.value}</h1>;
};

export default Headline;
```

And import it in another file:

```
// src/components/App.js

import React from 'react';

import Headline from './Headline.js';

const App = () => {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
};

export default App;
```

Note: If a Function Component is not defined, console log your exports and imports to get a

If you don't care about the component name by defining the variable, you can keep it as Anonymous Function Component when using a default export on the Function Component:

```
import React from 'react';

import Headline from './Headline.js';

export default () => {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
};
```

However, when doing it this way, React Dev Tools cannot identify the component because it has no display name. You may see an Unknown Component in your browser's developer tools.

REACT FUNCTION COMPONENT: REF

A React Ref should only be used in rare cases such as accessing/manipulating the DOM manually (e.g. focus element), animations, and integrating third-party DOM libraries (e.g. D3). If you have to use a Ref in a Function Component, you can define it within the component. In the following case, the input field will get focused after the component did mount:

```
import React, { useState, useEffect, useRef } from 'react';

const App = () => {
  const [greeting, setGreeting] = useState('Hello React!');
```

```
  return (
    <div>
      <h1>{greeting}</h1>

      <Input value={greeting} handleChange={handleChange} />
    </div>
  );
};

const Input = ({ value, handleChange }) => {
  const ref = useRef();

  useEffect(() => ref.current.focus(), []);

  return (
    <input
      type="text"
      value={value}
      onChange={handleChange}
      ref={ref}
    />
  );
};

export default App;
```

However, React Function Components cannot be given refs! If you try the following, the ref will be assigned to the component instance but not to the actual DOM node.

```
// Doesn't work!

import React, { useState, useEffect, useRef } from 'react';

const App = () => {
```

```
  const ref = useRef();

  useEffect(() => ref.current.focus(), []);

  return (
    <div>
      <h1>{greeting}</h1>

      <Input value={greeting} handleChange={handleChange} ref={ref} />
    </div>
  );
};

const Input = ({ value, handleChange, ref }) => (
  <input
    type="text"
    value={value}
    onChange={handleChange}
    ref={ref}
  />
);

export default App;
```

It's not recommended to pass a ref from a Parent Component to a Child Component and that's why the assumption has always been: React Function Components cannot have refs. However, if you need to pass a ref to a Function Component -- because you have to measure the size of a function component's DOM node, for example, or like in this case to focus an input field from the outside -- you can forward the ref:

```
// Does work!
```

```
  useRef,
  forwardRef,
} from 'react';

const App = () => {
  const [greeting, setGreeting] = useState('Hello React!');

  const handleChange = event => setGreeting(event.target.value);

  const ref = useRef();

  useEffect(() => ref.current.focus(), []);

  return (
    <div>
      <h1>{greeting}</h1>

      <Input value={greeting} handleChange={handleChange} ref={ref} />
    </div>
  );
};

const Input = forwardRef(({ value, handleChange }, ref) => (
  <input
    type="text"
    value={value}
    onChange={handleChange}
    ref={ref}
  />
));

export default App;
```

There are a few other things you may want to know about React Refs, so check out this article: How to use Ref in React or the official React documentation.

REACT FUNCTION COMPONENT: PROPTYPES

PropTypes can be used for React Class Components and Function Components the same way. Once you have defined your component, you can assign it PropTypes to validate the incoming props of a component:

```
import React from 'react';
import PropTypes from 'prop-types';

const App = () => {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
};

const Headline = ({ value }) => {
  return <h1>{value}</h1>;
};

Headline.propTypes = {
  value: PropTypes.string.isRequired,
};

export default App;
```

Note that you have to install the standalone React prop-types, because it has been removed from the React core library a while ago. If you want to learn more about PropTypes in React, check out the official documentation.

In addition, previously you have seen the usage of default props for a Function Component. For

```
import React from 'react';

const App = () => {
  const greeting = 'Hello Function Component!';

  return <Headline headline={greeting} />;
};

const Headline = ({ headline }) => {
  return <h1>{headline}</h1>;
};

Headline.defaultProps = {
  headline: 'Hello Component',
};

export default App;
```

Note that you can also use the default assignment when destructuring the value from the props in the function signature (e.g. `const Headline = ({ headline = 'Hello Component' }) =>`) or the `||` operator within the Function Component's body (e.g. `return <h1>{headline || 'Hello Component'}</h1>;`).

However, if you really want to go all-in with strongly typed components in React, you have to check out TypeScript which is briefly shown in the next section.

REACT FUNCTION COMPONENT: TYPESCRIPT

If you are looking for a type system for your React application, you should give TypeScript for React Components a chance. A strongly typed language like TypeScript comes with many

You may wonder: How much different would a React Function Component with TypeScript be? Check out the following typed component:

```
import React, { useState } from 'react';

const App = () => {
  const [greeting, setGreeting] = useState(
    'Hello Function Component!'
  );

  const handleChange = event => setGreeting(event.target.value);

  return (
    <Headline headline={greeting} onChangeHeadline={handleChange} />
  );
};

const Headline = ({
  headline,
  onChangeHeadline,
}: {
  headline: string,
  onChangeHeadline: Function,
}) => (
  <div>
    <h1>{headline}</h1>

    <input type="text" value={headline} onChange={onChangeHeadline} />
  </div>
);

export default App;
```

It only defines the incoming props as types. However, most of the time type inference just works

If you want to know how to get started with TypeScript in React, check out this comprehensive cheatsheet ranging from TypeScript setup to TypeScript recipes. It's well maintained and my go-to resource to learn more about it.

REACT FUNCTION COMPONENT VS CLASS COMPONENT

This section will not present you any performance benchmark for Class Components vs Functional Components, but a few words from my side about where React may go in the future.

Since React Hooks have been introduced in React, Function Components are not anymore behind Class Components feature-wise. You can have state, side-effects and lifecycle methods in React Function Components now. That's why I strongly believe React will move more towards Functional Components, because they are more lightweight than Class Components and offer a sophisticated API for reusable yet encapsulated logic with React Hooks.

For the sake of comparison, check out the implementation of the following Class Component vs Functional Component:

```
// Class Component

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

    this.state = {
      value: localStorage.getItem('myValueInLocalStorage') || '',
    };
```



```
  componentDidUpdate() {
    localStorage.setItem('myValueInLocalStorage', this.state.value);
  }

  onChange = event => {
    this.setState({ value: event.target.value });
  };

  render() {
    return (
      <div>
        <h1>Hello React ES6 Class Component!</h1>

        <input
          value={this.state.value}
          type="text"
          onChange={this.onChange}
        />

        <p>{this.state.value}</p>
      </div>
    );
  }
}

// Function Component

const App = () => {
  const [value, setValue] = React.useState(
    localStorage.getItem('myValueInLocalStorage') || '',
  );

  React.useEffect(() => {
    localStorage.setItem('myValueInLocalStorage', value);
  }, [value]);

  const onChange = event => setValue(event.target.value);
```

```
      <h1>Hello React Function Component!</h1>

      <input value={value} type="text" onChange={onChange} />

      <p>{value}</p>
    </div>
  );
};
```

If you are interested in moving from Class Components to Function Components, check out this guide: A migration path from React Class Components to Function Components with React Hooks. However, there is no need to panic because you don't have to migrate all your React components now. Maybe it's a better idea to start implementing your future components as Function Components instead.

. . .

The article has shown you almost everything you need to know to get started with React Function Components. If you want to dig deeper into testing React Components for instance, check out this in-depth guide: Testing React Components. Anyway, I hope there have been a couple of best practices for using Functional Components in React as well. Let me know if anything is missing!

Show Comments

KEEP READING ABOUT REACT

REACT HOOKS MIGRATION

React Hooks were introduced to React to make state and side-effects available in React Function Components. Before it was only possible to have these in React Class Components; but since React's way...

REACT HOOKS TUTORIAL

React Hooks were introduced at React Conf October 2018 as a way to use state and side-effects (see lifecycle methods in class components) in React function components. Whereas function components...

THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

✔ Tips and Tricks

✔ Access Tutorials, eBooks and Courses

✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

© Robin Wieruch