How to use SVG Icons as React Components

OCTOBER 30, 2020 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

I always struggled to use SVG in my React applications. Every time I searched about the topic online, I've found many ways on how to use SVG in React, but once I implemented the approaches, the success rates were very low. Today I want to give you a straightforward approach on how to use SVG icons as React components for your next React application.

Note: All icons used in this tutorial are from Flaticon. If you use icons from there, don't forget to attribute the authors/platform.

It's possible to have one folder in your React application that holds all your .svg files for your icons. From there, you can generate your React components manually/automatically. I will show you both approaches in the next two sections for creating icon components manually with your command line interface and npm scripts, but also for creating your icon components automatically with Webpack. The tool we are using is called SVGR which is widely used (e.g. create-react-app).

TABLE OF CONTENTS

REACT SVG ICON COMPONENTS FROM CLI

In this section, we will start by generating SVG icons manually for your React application. If you need a starter project, head over to this Webpack + Babel + React project and follow the installation instructions.

Next, put all your .svg icon files into a */assets* folder next to your *src/* folder. We don't want to have the assets mixed with our source code files, because we will generate JavaScript files based on them. These JavaScript files -- being React icon components -- are mixed with your other source code files then.

```
assets/
-- twitter.svg
-- facebook.svg
-- github.svg
src/
-- index.js
-- App.js
```

Now, create an empty *src/Icons/* folder for all your generated React icon components:

```
assets/
-- twitter.svg
-- facebook.svg
-- github.svg
src/
-- index.js
-- App.js
-- Icons/
```

The desired outcome would be to use the React icon components in our *src/App.js* component:

```
import React from 'react';

import TwitterIcon from './Icons/Twitter.js';
import FacebookIcon from './Icons/Facebook.js';
import GithubIcon from './Icons/Github.js';

const App = () => (
  <div>
    <ul>
      <li>
        <TwitterIcon width="40px" height="40px" />
        <a href="https://twitter.com/rwieruch">Twitter</a>
      </li>
      <li>
        <FacebookIcon width="40px" height="40px" />
        <a href="https://www.facebook.com/rwieruch/">Facebook</a>
      </li>
      <li>
        <GithubIcon width="40px" height="40px" />
        <a href="https://github.com/rwieruch">Github</a>
      </li>
    </ul>
  </div>
);

export default App;
```

However, this doesn't work yet because the *src/Icons/* folder is empty. There are no icon components yet. In the next step, the *assets/* folder will act as *source* folder and the *src/Icons/* as *target* folder. We will add a new npm script to our *package.json* file which will generate the React icon components:

```
{
  ...
  "main": "index.js",
  "scripts": {
    "svgr": "svgr -d src/Icons/ assets/",
    "start": "webpack serve --config ./webpack.config.js --mode development"
  },
  "keywords": [],
  ...
}
```

Last but not least, install the SVGR CLI package on the command line:

```
npm install @svgr/cli --save-dev
```

Now, after having everything set up properly, you can execute your new npm script with `npm run svgr` on the command line. Reading the command line output, you can see that new JavaScript files are being generated from your svg files. After the command terminates, you should be able to see the svg icons rendered as React components when starting your application. You can also check your *src/Icons* folder to see all generated React icon components. They take props as arguments as well, which makes it possible for us to define their height and width.

That's all it needs to generate React components from SVGs. Every time you have a new SVG file or adjust one of your existing SVG files, you can the `npm run svgr` command again.

REACT SVG ICON COMPONENTS FROM WEBPACK

Running the SVGR script every time to update your SVG icons is not the best solution though. What about integrating the whole process in your Webpack configuration? You should start out with an empty *src/Icons* folder. Afterward, move all your SVG files to this folder and remove the *assets/* folder. Your folder structure should look like the following:

```
src/
-- index.js
-- App.js
-- Icons/
---- twitter.svg
---- facebook.svg
---- github.svg
```

Your App component imports SVG files rather than JavaScript files now:

```
import React from 'react';

import TwitterIcon from './Icons/Twitter.svg';
import FacebookIcon from './Icons/Facebook.svg';
import GithubIcon from './Icons/Github.svg';

const App = () => (
  <div>
    <ul>
      <li>
        <TwitterIcon width="40px" height="40px" />
        <a href="https://twitter.com/rwieruch">Twitter</a>
      </li>
      <li>
        <FacebookIcon width="40px" height="40px" />
        <a href="https://www.facebook.com/rwieruch/">Facebook</a>
      </li>
      <li>
```

```
          <GithubIcon width="40px" height="40px" />
          <a href="https://github.com/rwieruch">Github</a>
        </li>
      </ul>
    </div>
);

export default App;
```

Starting your application wouldn't work, because we cannot simply import SVG files this way. Fortunately, we can make Webpack doing the work for us implicitly with every application start. Let's add the following configuration to our *webpack.config.js* file:

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: ['babel-loader'],
      },
      {
        test: /\.svg$/,
        use: ['@svgr/webpack'],
      },
    ],
  },
  ...
};
```

Afterward, install the necessary Webpack package for SVGR:

```
npm install @svgr/webpack --save-dev
```

Once you start your application, Webpack is doing its thing and you don't need to worry about your SVGs anymore. You can put your SVG files anywhere in your *src/* folder and import them wherever you need them as React components. There is no need anymore for the SVGR npm script in your *package.json* file which we have implemented in the previous section.

Alternative: react-svg-loader

If you are using Webpack, you can also use a simplified SVG loader instead of SVGR. For instance, react-svg-loader can be used within your Webpack configuration. Note that it replaces SVGR:

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: ['babel-loader'],
      },
      {
        loader: 'react-svg-loader',
        options: {
          jsx: true // true outputs JSX tags
        }
      }
    ],
  },
  ...
};
```

Also you need to install it:

```
npm install react-svg-loader --save-dev
```

Afterward, you can import SVG files the same way as React components as you did before with SVGR. It can be seen as a lightweight alternative to SVGR.

SVGR TEMPLATES FOR ADVANCED SVGS

When I worked with my last client on their React application, I had the problem of dealing with SVG icons which had only partially the viewBox attribute. Since this attribute is needed for giving your SVG icons a size, I had to find a way around to introduce this attribute whenever it wasn't present for an icon. Now I could go through every SVG icon to fix this issue, however, dealing with more than 500 icons doesn't make this a comfortable task. Let me show how I dealt with it by using SVGR templates instead.

The default SVGR template in your *webpack.config.js* file looks like the following:

```
...

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: ['babel-loader'],
      },
      {
        test: /\.svg$/,
        use: [
          {
            loader: '@svgr/webpack',
            options: {
              template: (
                { template },
                opts,
                { imports, componentName, props, jsx, exports }
              ) => template.ast`
                ${imports}

                const ${componentName} = (${props}) => {
                  return ${jsx};
                };

                export default ${componentName};
              `,
            },
          },
        ],
      },
    ],
  },
  ...
};
```

By having this template at your disposal, you can change the code which is generated from the SVG file. Let's say we want to fill all our icons with a blue color. We just extend the props object with a fill attribute:

```
...

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: ['babel-loader'],
      },
      {
        test: /\.svg$/,
        use: [
          {
            loader: '@svgr/webpack',
            options: {
              template: (
                { template },
                opts,
                { imports, componentName, props, jsx, exports }
              ) => template.ast`
                ${imports}

                const ${componentName} = (${props}) => {
                  props = { ...props, fill: 'blue' };

                  return ${jsx};
                };

                export default ${componentName};
              `,
            },
          },
        ],
      },
    ],
  },
  ...
};
```

This should work to give all icons a blue fill attribute. However, simple use cases like this are already provided by SVGR itself. Just check out their documentation on how to add/replace/remove attribute from SVGs.

SVGR with custom viewBox attribute

In our case, we wanted to compute the viewBox attribute for every SVG icon where the attribute isn't present. First, remove the viewBox attribute from one of your SVGs to see that it's not rendered properly anymore. After confirming the bug, we will try to fix it by using the introduced SVGR template and an external React Hook:

```
import React from 'react';

const useWithViewbox = ref => {
  React.useLayoutEffect(() => {
    if (
      ref.current !== null &&
      // only if there is no viewBox attribute
      !ref.current.getAttribute('viewBox') &&
      // only if not test (JSDOM)
      // https://github.com/jsdom/jsdom/issues/1423
      ref.current.getBBox &&
      // only if rendered
      // https://stackoverflow.com/questions/45184101/error-ns-error-failure-
      ref.current.getBBox().width &&
      ref.current.getBBox().height
    ) {
      const box = ref.current.getBBox();

      ref.current.setAttribute(
        'viewBox',
        [box.x, box.y, box.width, box.height].join(' ')
      );
    }
  });
};

export default useWithViewbox;
```

The React hook only needs a reference (ref) to the SVG components in order to set the viewBox attribute. The measurements for the viewBox attribute are computed based on the rendered icon. If the icon hasn't been rendered or the viewBox attribute is already present, we do nothing.

The hook should be somewhere available not far away from our *src/Icons/* folder:

```
src/
-- index.js
-- App.js
-- useWithViewbox.js
-- Icons/
---- twitter.svg
---- facebook.svg
---- github.svg
```

Now, we can use the hook for our SVG template in the *webpack.config.js* file:

```
...

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      ...
      {
        test: /\.svg$/,
        use: [
          {
            loader: '@svgr/webpack',
            options: {
              template: (
                { template },
                opts,
                { imports, componentName, props, jsx, exports }
              ) => template.ast`
                ${imports}
                import useWithViewbox from '../useWithViewbox';

                const ${componentName} = (${props}) => {
                  const ref = React.useRef();

                  useWithViewbox(ref);

                  props = { ...props, ref };

                  return ${jsx};
                };

                export default ${componentName};
              `,
            },
          },
        ],
      },
    ],
  },
  ...
};
```

Having this in place, SVGR's template feature will add the custom hook to every generated icon component. The hook only runs for icon components which have no viewBox attribute though. If you run your application again, you should see all icon components rendered properly, even though you may have removed the viewBox attribute from one of them.

. . .

In the end, I hope this walkthrough has helped you to get started with SVG icons in React by using SVGR with your command line/npm scripts or Webpack. The finished application using the Webpack approach and React can be found in this GitHub repository. If you run into any bugs, let me know in the comments. Otherwise I am happy to heard about your special use cases which fall into the category of my missing viewBox bug. Let me know about these cases in the comments.

Show Comments

KEEP READING ABOUT REACT

HOW TO USECONTEXT IN REACT?

React's Function Components come with React Hooks these days. Not only can React Hooks be used for State in React but also for using React's Context in a more convenient way. This tutorial shows...

HOW TO USE REACT SVG PATTERNS AS BACKGROUNDS

Recently I was confronted with SVG in React. I had to use a logo and wanted to use playful SVG background patterns in React. By using SVG, I avoided to use other image formats which would add...

GET THE BOOK ›

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✓ Join 50.000+ Developers

✓ Learn Web Development with JavaScript

✓ Tips and Tricks

✓ Access Tutorials, eBooks and Courses

✓ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO	ABOUT
Online Courses	About me
Open Source	What I use

Tutorials

How to work with me

How to support me

Contact Me Privacy & Terms