The Best Way to Import SVGs in React

The how and the why

Mohamed Lamine Allal Follow
Feb 4, 2019 · 6 min read ★

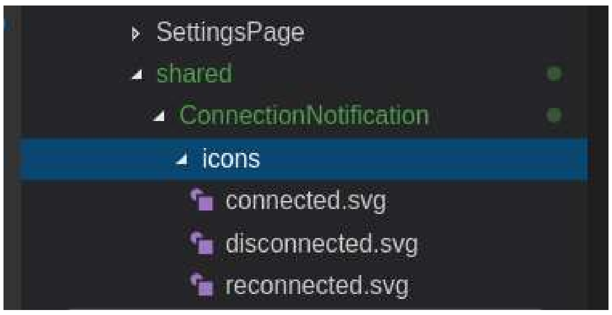
Screenshot by the author

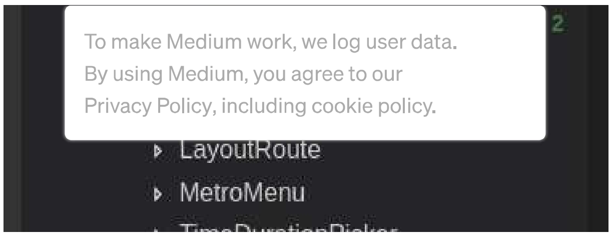
Who doesn't appreciate the coolness that SVG brings? SVG support in HTML 5 is by all means a great technology that offers a lot. And in all its simplest forms, we appreciate how our icons or graphics can scale without losing their sharpness. Also, we like to be able to control and play with the details that form our graphics, to style them, animate them, and so on.

In this article, I will show the two methods to import SVG assets into React components. I like to call them the *in-source assets* as they will be situated in our source (create-react-app by default doesn't allow importing from outside `src` .That's restricted unless you eject and change the config. (That's for a good reason, which I'll treat in another article.)

I will show two methods, the old one and a kind of new one that comes with create-react-app V2. I'll explain the motivation and the why.

Note: If you're in a hurry, scroll down to the next heading.





To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

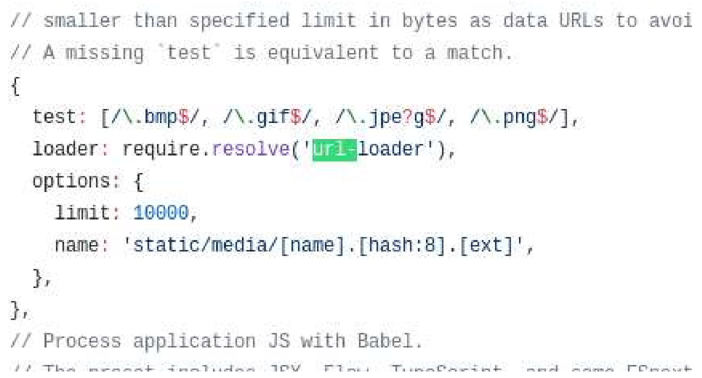
Image source: Author

First of all, due to webpack and loaders `url-loader` and `file-loader` , importing images and SVG is possible. This was already leveraged by default from the start.

Note: `url-loader` creates a data URL from images (base-64 data URL, embedded in the HTML directly, for a fast load and no fetch after page load). That's unless they are greater than a certain limit (in create-react-app, it's 10,000), in which case it falls back to **`file-loader`** , which will output the imported file into the configured output. (This is the default behavior if no fallback is defined, and so it is with create-react-app). And the output in create-react-app is the build folder in production and the development server in development.

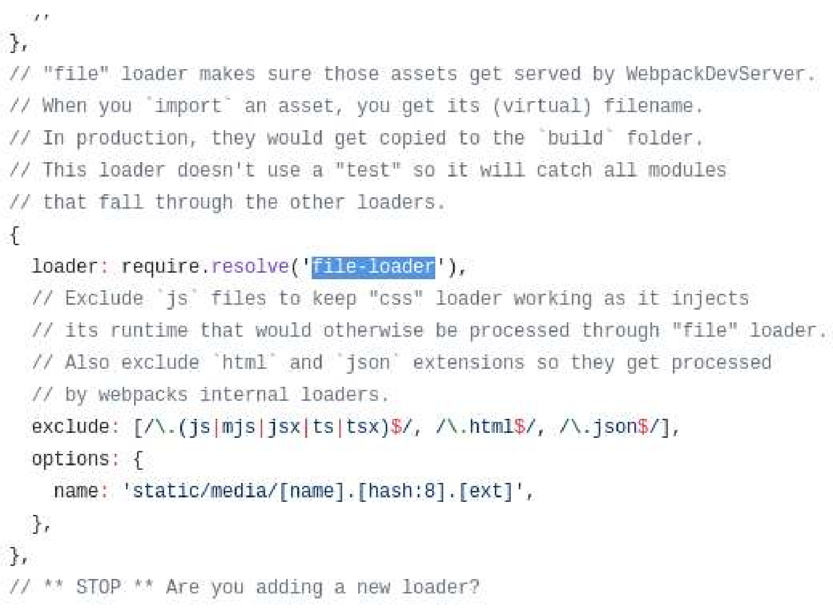
The code below is captured from create-react-app `webpack.config.js` :

`url-loader`



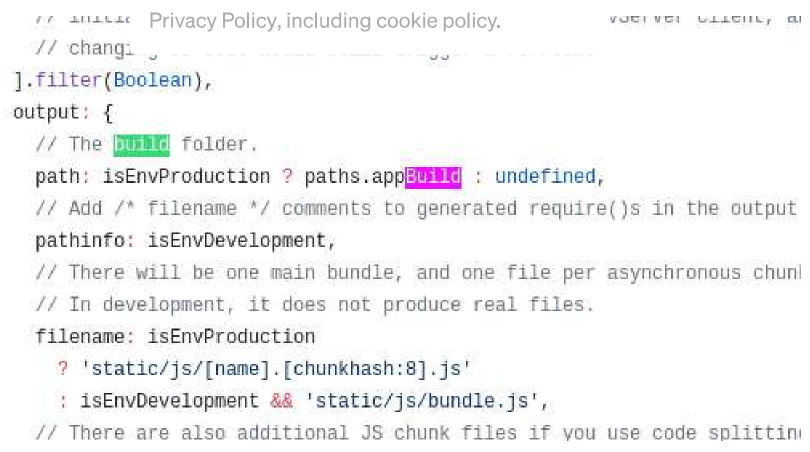
```
// smaller than specified limit in bytes as data URLs to avoi
// A missing `test` is equivalent to a match.
{
  test: [/\.bmp$/, /\.gif$/, /\.jpe?g$/, /\.png$/],
  loader: require.resolve('url-loader'),
  options: {
    limit: 10000,
    name: 'static/media/[name].[hash:8].[ext]',
  },
},
// Process application JS with Babel.
```

`file-loader`



```
},
// "file" loader makes sure those assets get served by WebpackDevServer.
// When you `import` an asset, you get its (virtual) filename.
// In production, they would get copied to the `build` folder.
// This loader doesn't use a "test" so it will catch all modules
// that fall through the other loaders.
{
  loader: require.resolve('file-loader'),
  // Exclude `js` files to keep "css" loader working as it injects
  // its runtime that would otherwise be processed through "file" loader.
  // Also exclude `html` and `json` extensions so they get processed
  // by webpacks internal loaders.
  exclude: [/\.(js|mjs|jsx|ts|tsx)$/, /\.html$/, /\.json$/],
  options: {
    name: 'static/media/[name].[hash:8].[ext]',
  },
},
// ** STOP ** Are you adding a new loader?
```

output



```
  // changi
].filter(Boolean),
output: {
  // The build folder.
  path: isEnvProduction ? paths.appBuild : undefined,
  // Add /* filename */ comments to generated require()s in the output
  pathinfo: isEnvDevelopment,
  // There will be one main bundle, and one file per asynchronous chunl
  // In development, it does not produce real files.
  filename: isEnvProduction
    ? 'static/js/[name].[chunkhash:8].js'
    : isEnvDevelopment && 'static/js/bundle.js',
  // There are also additional JS chunk files if you use code splittin
```

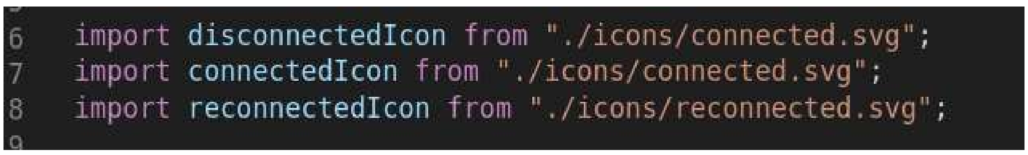
This gives a good idea of how it happens. (If you want to know more, read about webpack loaders and system).

Now back to how we import. Let's start with the old way. (Make sure to read to the end. If you're in a hurry, jump to the next headline.)

Import Images or SVG Directly as a URL

I will take an example from an actual project. I needed to build a component to highlight or notify about connection and disconnection in a real-time application.

Back to the line. To import, we go with the following:

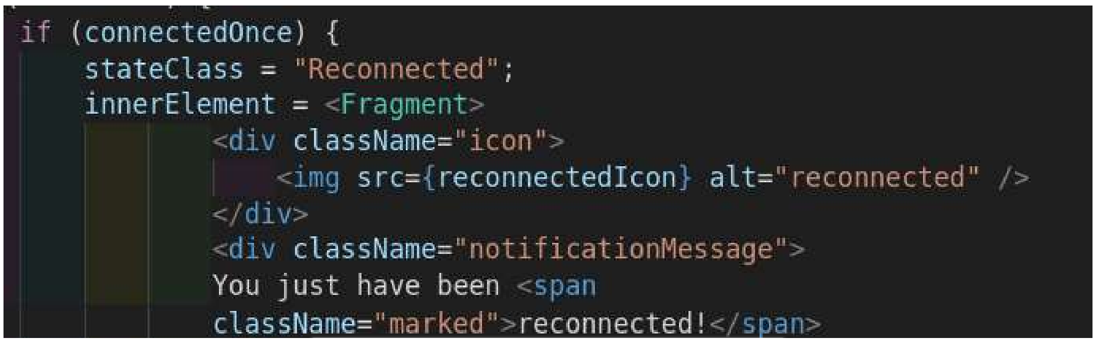


```
import disconnectedIcon from "./icons/connected.svg";
import connectedIcon from "./icons/connected.svg";
import reconnectedIcon from "./icons/reconnected.svg";
```

```
import myIcon from './relative/path/to/icon.svg'
```

We use the normal import syntax, and we import our SVG assets, thanks to the `url-loader` for webpack. Our variable will hold a URL to the SVG icon.

We use it like this:



```
if (connectedOnce) {
    stateClass = "Reconnected";
    innerElement = <Fragment>
            <div className="icon">
                <img src={reconnectedIcon} alt="reconnected" />
            </div>
            <div className="notificationMessage">
            You just have been <span
            className="marked">reconnected!</span>
```

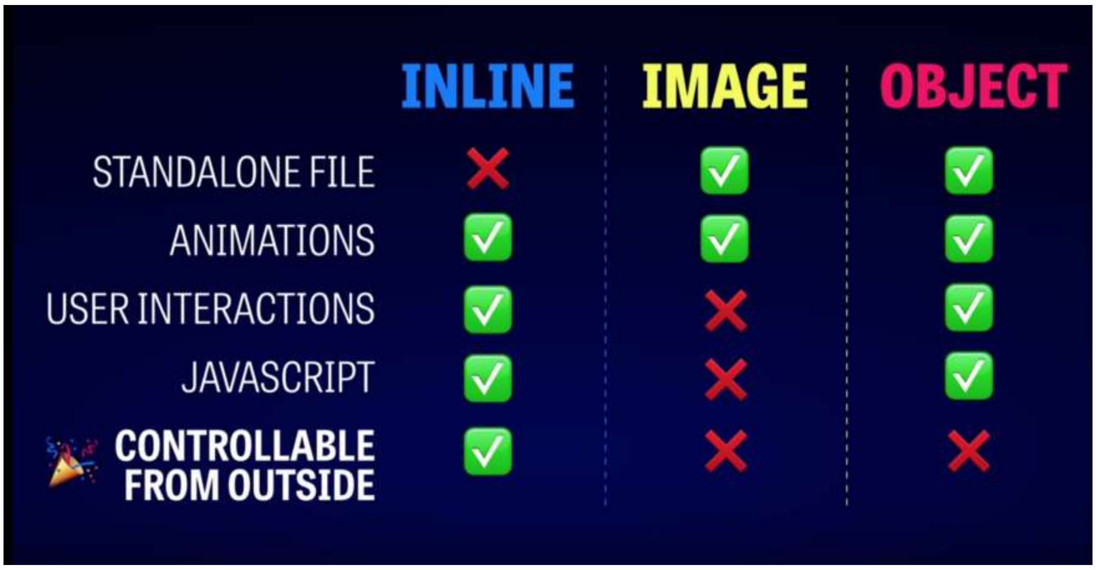
```
</div>
```

```
<img src={myIcon} alt="icon" />
```

We come to the line with the problem, why this may not be ok or why we may need something else:

SVG as an image doesn’t offer any control. We will not be able to style it as we want and neither animate nor control the inner SVG components.

Here’s an image that illustrates the different ways to use SVG with HTML and how they compare in terms of features.



	INLINE	IMAGE	OBJECT
STANDALONE FILE	❌	✅	✅
ANIMATIONS	✅	✅	✅
USER INTERACTIONS	✅	❌	✅
JAVASCRIPT	✅	❌	✅
🎉 CONTROLLABLE FROM OUTSIDE	✅	❌	❌

Image source: Author

We can see that the inline method is the most suitable. So you might just think the way to go is to make components that will render our inline SVG. And that’s what the second method is about.

Import SVG as a React Component (Inline SVG)

Due to the disability of the image method, an issue was raised. create-react-app v2 ended up including a solution for that, as it did with a bunch of other features. Our life was just made even easier. We need to thank all the great guys out there (thank you, folks). You can look at this thread to see when the issue was filled https://github.com/facebook/create-react-app/issues/1388.

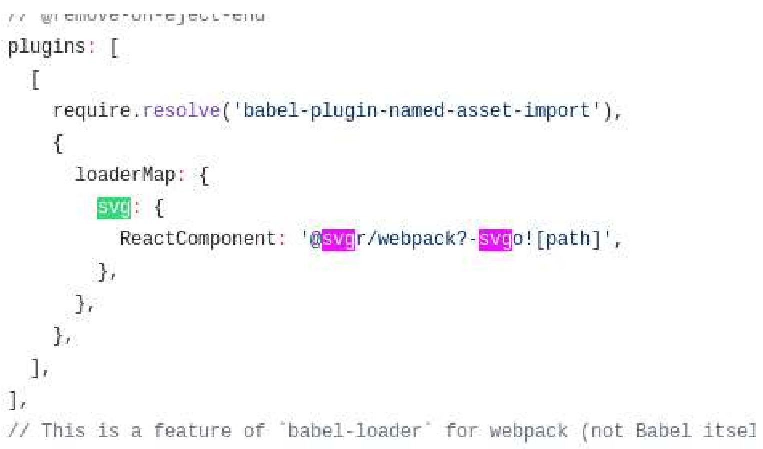
This time a webpack config is leveraged for us, using the svgr loader for webpack, which you can check out here https://www.npmjs.com/package/@svgr/webpack).

Also, you can check out the svgr repo https://github.com/smooth-code/svgr to learn more about svgr.

There is a playground th

svgr takes an SVG file an that renders an inline SVG.

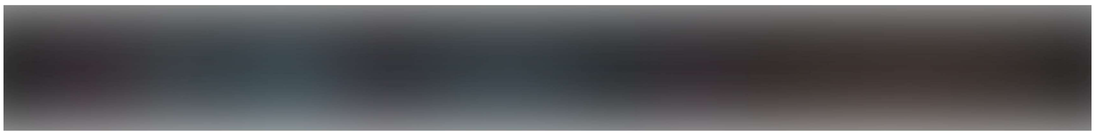
Here is create-react-app webpack config:



```
// @remove-on-eject-end
plugins: [
  [
    require.resolve('babel-plugin-named-asset-import'),
    {
      loaderMap: {
        svg: {
          ReactComponent: '@svgr/webpack?-svgo![path]',
        },
      },
    },
  ],
],
// This is a feature of `babel-loader` for webpack (not Babel itsel
```

This is the story. Let’s see how we do the import.

Back to our example — let’s make some modifications.



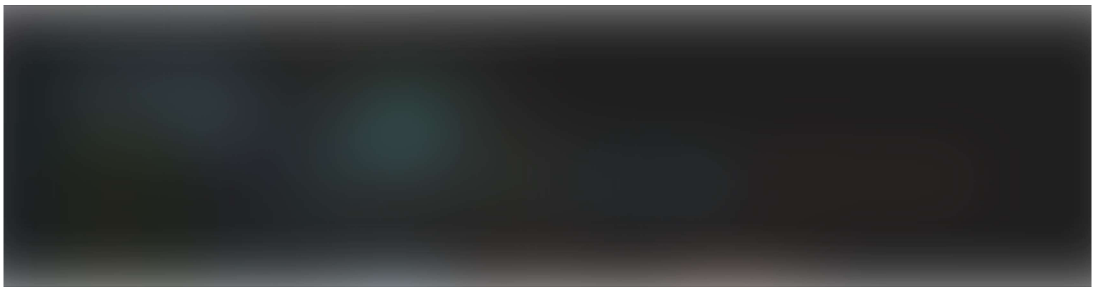
```
import { ReactComponent as MyIcon } from "./icon.svg"
```

We can import both versions, which doesn’t really make sense, but anyway, let’s see it.

```
import myIconUrl, { ReactComponent as MyIcon } from "./icon.svg"
```

Make sure to import `ReactComponent` from the model as a specific component name so that it makes sense. And make sure you respect that it starts with a capital letter. Otherwise JSX will be transpiled to a string literal and not a variable.

Here the component in place:

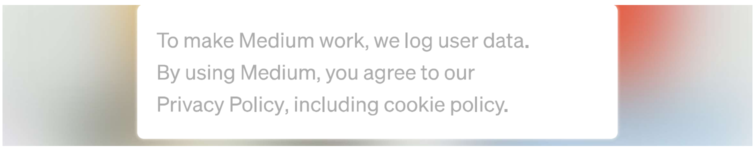


```
<MyIcon />
<MyIcon className='                                  t="icon" />
```

Now we need to see how both methods render.

I made a refactor to my `ConnectionNotification` component, which, by the way, I will share and publish once it's all good to go. (It handles showing when it's disconnected. At the start, it shows if it's connected, or, if not, it will show "disconnected" but only after a wait time passes. Also "connected" will show immediately and hide after a timeout. In short it handles the notification feature, and well. And it can be costumed with style, and icons, text, etc.)

The component above renders to the following inline SVG, as you can see:



While the image version:

Will render:

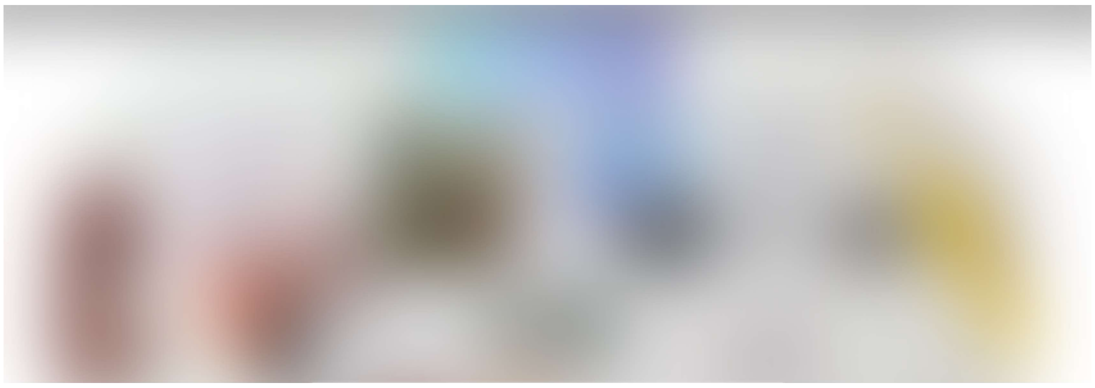
Note that it makes no difference in my use case from a style viewpoint. However, know that inline SVG will load faster, which alone is a good reason. If I want, I can make some animations on the inner SVG components.

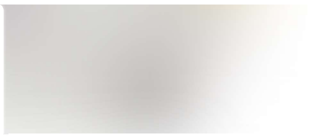
Now that you have an inline SVG, you can work as usual and have all the control that you need. By far this is the best way to go.

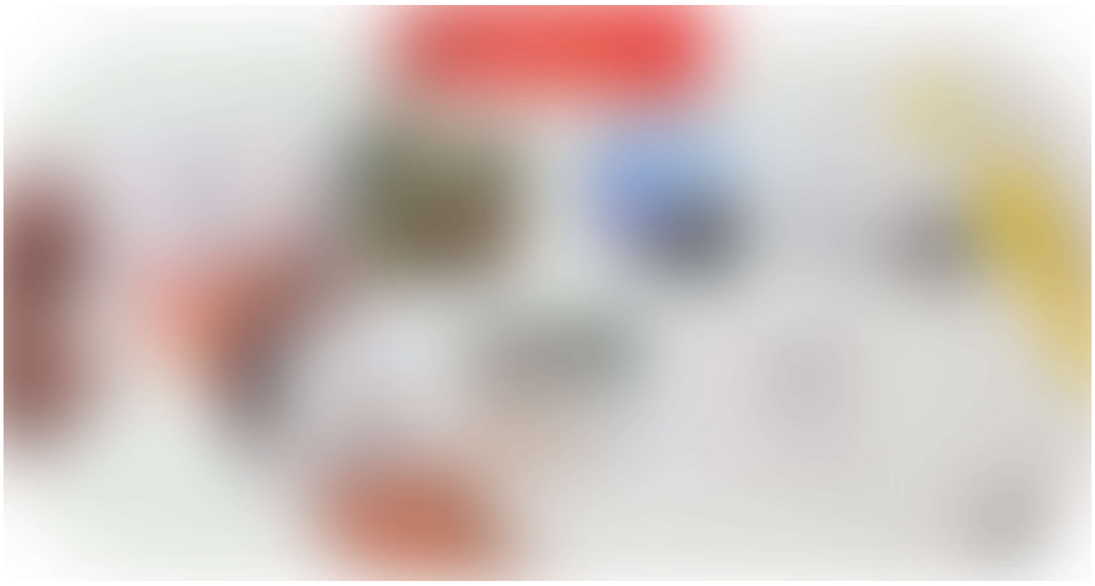
Note: Here a link for a gist that shows one of the SVG sprite files I used.

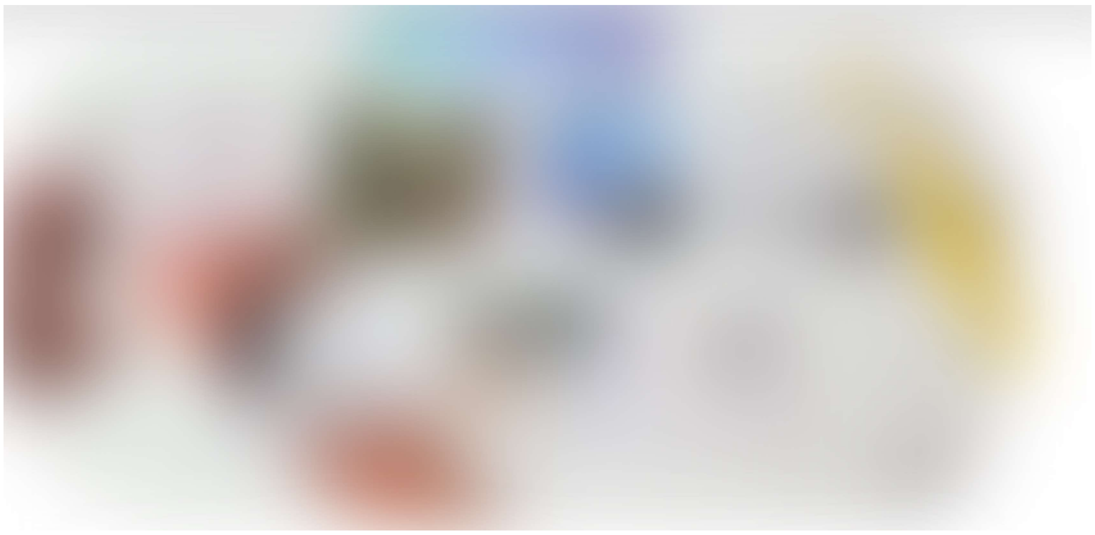
https://gist.github.com/MohamedLamineAllal/a1bcca5c4e7d12defddfceabc2a2734d?short_path=7802baa

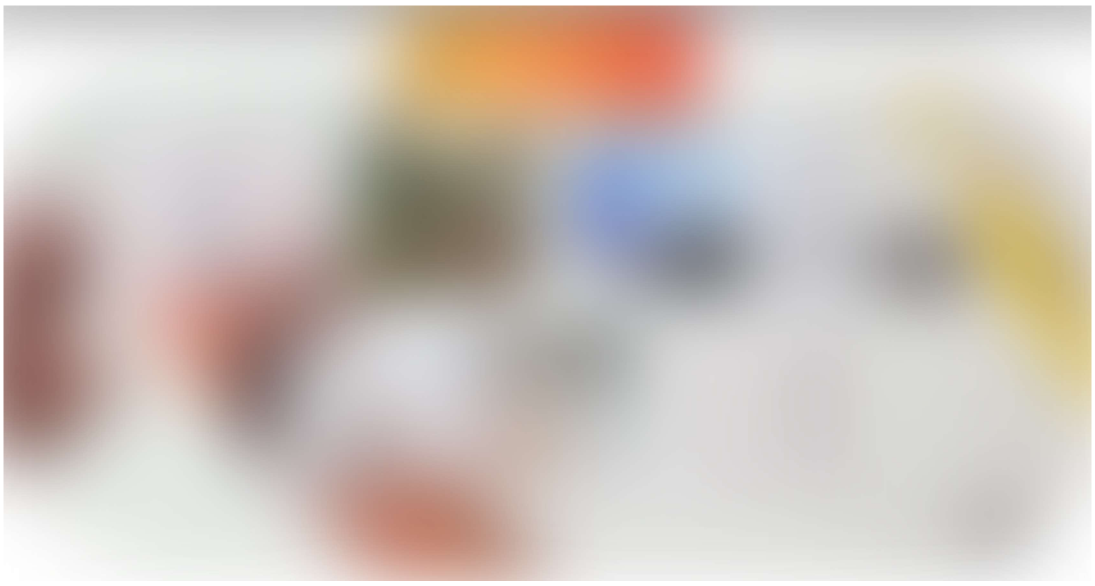
And just to show it, here is my final connection component rendered:











I hope you love it.

Sign up for prograr

By Better Programming

A monthly newsletter covering the best programming articles published across Medium. Code tutorials, advice, career opportunities, and more! Take a look.

Get this newsletter

Programming React JavaScript Reactjs Nodejs