Migrating from gatsby-image to gatsby-plugin-image

This document is provided to help users of `gatsby-image` migrate to using `gatsby-plugin-image`.

Note that you can use both packages at the same time. You may need to in situations where you're using a `gatsby-image`-compatible CMS as a data source, and that CMS plugin has not yet updated to a version compatible with the new API.

If you're looking for other documentation on the new plugin, see:

- How-To Guide: Using the beta Gatsby Image plugin
- Reference Guide: Gatsby Image plugin

What changed?

New Syntax

The new plugin requires significant syntax changes. We've provided a codemod to help you migrate, but this section will explain the changes.

Import change Previously, `GatsbyImage` was the default export from `gatsby-image`. With `gatsby-plugin-image`, the component is now a named export.

```
// import Img from "gatsby-image"
import { GatsbyImage } from "gatsby-plugin-image"
```

GraphQL changes This is one of the bigger changes. There are no more fragments to use. Instead, things like `layout` and `format` are passed as arguments to the resolver.

This is the previous syntax.

```
import { graphql } from "gatsby"

export const query = graphql`
  {
    file(relativePath: { eq: "images/example.jpg" }) {
      childImageSharp {
        fixed {
          ...GatsbyImageSharpFixed
        }
      }
    }
  }
`
```

The new syntax looks like this. The fragment is removed in favor of `gatsbyImageData`, which is used for all images. Previous configuration options - such as `fixed` vs. `fluid`, WebP generation, and placeholder type - are passed as arguments to the resolver.

```
import { graphql } from "gatsby"

export const query = graphql`
  {
    file(relativePath: { eq: "images/example.jpg" }) {
      childImageSharp {
        gatsbyImageData(layout: FIXED)
      }
    }
  }
`
```

Other changes to the available argument structure are in the section on API changes.

Component changes The last change is to the JSX component. The import name is potentially different, and the query result is also different.

```
// import Img from "gatsby-image"
import { GatsbyImage } from "gatsby-plugin-image"

const HomePage = ({ data }) => {
  return (
    // <Img fixed={data.file.childImageSharp.fixed} />
    <GatsbyImage image={data.file.childImageSharp.gatsbyImageData} />
  )
}
```

API Changes

In addition to the syntax changes for using `gatsby-plugin-image`, there are also changes to the API that affect the resolver arguments (and the new `StaticImage` component).

fluid The `fluid` image type has been deprecated in favor of two alternatives.

The first is an image type called `fullWidth`. This image is designed to be used for things like hero images that span the full width of the screen, and generates image sizes accordingly. Instead of passing something like `maxWidth`, it takes an array called `breakpoints` that will generate images designed for those screen sizes. In most cases you will not need to provide these, as there are default values that work for most sites. Note that like the old `fluid` layout, `fullWidth` images will expand to fit the width of their container, even if that width is larger than the source image.

The second is a responsive image type called `constrained` that will shrink with its container and generates smaller images, but nothing larger than the original image source size. Additionally, you can pass `width` and/or `height` to limit the displayed image to that size. In this case, larger images may be generated for high-density screens.

maxWidth `maxWidth` and `maxHeight` are deprecated for all image types. For `constrained` and `fixed` images, use `width` and `height`. For `fullWidth` images, look at the `breakpoints` option.

aspectRatio `aspectRatio` is a new argument that takes a number (or fraction). If you pass it without also passing in `width` or `height`, it will default to using the source image width and then adjusting the height to the correct aspect ratio. Alternatively, you can pass your own `width` or `height` and it will set the other dimension. Passing both `height` and `width` will cause `aspectRatio` to be ignored in favor of the inferred aspect ratio.

formats Previously, images generated their own type by default, e.g. JPG, PNG, etc. You could also generate WebP images when using the appropriate fragment. This is now controlled using the `formats` argument. This field takes an array, `[AUTO, WEBP]` by default, where `AUTO` means the same format as the source image.`AVIF` is now also a valid format.

Options nested inside objects Previously, transformations like `grayscale` and quality options such as `pngQuality` were top level query arguments. This has changed.

`grayscale` now exists within the `transformOptions` argument, and `pngQuality` becomes `quality` inside `pngOptions`. The `traceSVG` object is now `tracedSVGOptions`. See the `gatsby-plugin-image` Reference Guide for specifics.

Breaking changes

Due to the changes to `gatsby-plugin-image`, there is some functionality that is no longer supported.

1. `GatsbyImage` is no longer a class component and therefore cannot be extended. You can use composition instead.
2. `fluid` images no longer exist, and the `fullWidth` replacement does not take `maxWidth` or `maxHeight`.
3. The art direction API has changed, see the `gatsby-plugin-image` reference guide for specifics.
4. The component no longer takes a decomposed object, and the following code is not valid. You should avoid accessing or changing the contents of the `gatsbyImageData` object, as it is not considered to be a public API, so can be changed without notice.

   ```
   // THIS IS NOT VALID
   <GatsbyImage image={{ src: example.src, srcSet: ``, width: 100 }} />
   ```

How to Migrate

1. Install and update dependencies.

   ```
   npm install gatsby-plugin-image gatsby-plugin-sharp gatsby-transformer-sharp
   ```

2. Configure plugins.

   ```
   module.exports = {
     plugins: [
       `gatsby-plugin-image`,
       `gatsby-plugin-sharp`,
       `gatsby-transformer-sharp`,
     ],
   }
   ```

3. Run the codemod.

   Note that if you need to do a partial migration, e.g. because you're using a CMS that doesn't yet support the new plugin alongside local image files, you'll want to make use of the `optional-path` to run against individual files.

   ```
   npx gatsby-codemods gatsby-plugin-image <optional-path>
   ```

   Without an `optional-path`, the codemod will run against all the files in your current directory, so running it in root is recommended. It will ignore `node_modules`, `.cache`, and `public` automatically. It will also respect any local Babel configuration in your project. The codemod is designed to run against files with the extensions `.ts`, `.js`, `.tsx`, and `.jsx`. If this does not cover your project, or you require other customizations, see the section on using `jscodeshift`.

   Due to the API changes, the codemod is not a pure 1:1 mapping. There are some changes introduced.

- Fluid images will map to either `fullWidth` or `constrained` images. This decision is made based on the existence of `maxWidth` and its value. If `maxWidth` does not exist, it will be a `fullWidth` image. If it does, and the `maxWidth` is less than 1000, it will be a `constrained` image, otherwise a `fullWidth` image. `fullWidth` images do not retain their `maxWidth` or `maxHeight` fields; `constrained` images do, as `width` and `height`.
- All images will generate WebP.

The codemod will output warnings in a number of different scenarios and point you to the file in question so you can inspect the changes manually.

4. Consider manual changes.

- For images using static query, you should move to use the `StaticImage` component instead. This component takes `src`, which can be a remote image URL or a relative path to an image. Make sure you've installed `gatsby-source-filesystem` if you're going to use this component.

```
import { StaticImage } from "gatsby-plugin-image"

const HomePage = () => (
      <StaticImage src="./example.jpg" alt="please include an alt" />
  )
}
```

- You may also consider refactoring code to make use of the `getImage` helper function.

```
import { getImage, GatsbyImage } from "gatsby-plugin-image"

const HomePage = ({ data }) => {
  const image = getImage(data.file)
  return (
    <>
      <GatsbyImage image={image} alt="please include an alt" />
    </>
  )
}
```

- Finally, if you were previously using `src`, e.g. for an SEO component, you'll want to use the `getSrc` helper function as the internal structure of the return object has changed.

```
import { getSrc } from "gatsby-plugin-image"

const HomePage = ({ data }) => {
  const imagePath = getSrc(data.file)
  return (
    <>
      <SEO imageSrc={imagePath} />
    </>
  )
}
```

jscodeshift

This section is for people who need to run the codemod with extra flags exposed by `jscodeshift`, e.g. to transform file types other than those with extensions `.js`, `.ts`, `.tsx`, or `.jsx`.

1. Install `jscodeshift`.

```
npm install --global jscodeshift
```

2. Install the codemods package in your project.

```
npm install gatsby-codemods
```

3. Run the codemod using `jscodeshift`.

```
jscodeshift -t node_modules/gatsby-codemods/transforms/gatsby-plugin-image.js .
```

See the jscodeshift docs for all the available flags.