Gatsby Image API

This document talks about the deprecated image plugin. Please check out how to work with the new gatsby-plugin-image

Part of what makes Gatsby sites so fast is its recommended approach to handling images. `gatsby-image` is a React component designed to work seamlessly with Gatsby's native image processing capabilities powered by GraphQL and gatsby-plugin-sharp to easily and completely optimize image loading for your sites.

> *Note: gatsby-image is **not** a drop-in replacement for `<img />`. It's optimized for responsive fixed width/height images and images that stretch the full-width of a container. There are also other ways to work with images in Gatsby that don't require GraphQL.*

Demo: https://using-gatsby-image.gatsbyjs.org/

Setting up Gatsby Image

To start working with Gatsby Image, install the `gatsby-image` package along with necessary plugins `gatsby-transformer-sharp` and `gatsby-plugin-sharp`. Reference the packages in your `gatsby-config.js` file. You can also provide additional options to `gatsby-plugin-sharp` in your config file.

A common way to source images is to install and use `gatsby-source-filesystem` to connect your local files, but other source plugins can be used as well, such as `gatsby-source-contentful`, `gatsby-source-datocms` and `gatsby-source-sanity`.

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

```
module.exports = {
  plugins: [
    `gatsby-transformer-sharp`,
    `gatsby-plugin-sharp`,
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        path: `${__dirname}/src/data/`,
      },
    },
```

```
  ],
}
```

For in-depth install instructions, check out the docs on Using Gatsby Image.

Gatsby image starts with a query

To feed file data into Gatsby Image, set up a GraphQL query and either pass it into a component as props or write it directly in the component. One technique is to leverage the `useStaticQuery` hook.

Common GraphQL queries for sourcing images include `file` from gatsby-source-filesystem, and both `imageSharp` and `allImageSharp` from gatsby-plugin-sharp, but ultimately the options available to you will depend on your content sources.

> Note: you can also use GraphQL aliases for querying multiple images of the same type.

See below for code examples of queries and how to use them in components.

Types of images with `gatsby-image`

Gatsby image objects are created through GraphQL methods. There are two types of image optimizations available, *fixed* and *fluid*, which create multiple image sizes (1x, 1.5x, etc.). There is also the *resize* method, which returns a single image.

Images with a *fixed* width and height

Automatically create images for different resolutions at a set width or height — Gatsby creates responsive images for 1x, 1.5x, and 2x pixel densities using the `<picture>` element.

Once you've queried for a `fixed` image to retrieve its data, you can pass that data into the `Img` component:

```
import { useStaticQuery, graphql } from "gatsby"
import Img from "gatsby-image"

export default function Image() {
  const data = useStaticQuery(graphql`
    query {
      file(relativePath: { eq: "images/default.jpg" }) {
        childImageSharp {
          # Specify a fixed image and fragment.
          # The default width is 400 pixels
          // highlight-start
          fixed {
            ...GatsbyImageSharpFixed
```

```
          }
          // highlight-end
        }
      }
    }
  `)
  return (
    <div>
      <h1>Hello gatsby-image</h1>
      <Img
        fixed={data.file.childImageSharp.fixed} {/* highlight-line */}
        alt="Gatsby Docs are awesome"
      />
    </div>
  )
}
```

Fixed image query parameters In a query, you can specify options for fixed images.

- `width` (int, default: 400)
- `height` (int)
- `quality` (int, default: 50)

Returns

- `base64` (string)
- `aspectRatio` (float)
- `width` (float)
- `height` (float)
- `src` (string)
- `srcSet` (string)

This is where fragments like `GatsbyImageSharpFixed` come in handy, as they'll return all the above items in one line without having to type them all out:

```
file(relativePath: { eq: "images/default.jpg" }) {
  childImageSharp {
    // highlight-start
    fixed(width: 400, height: 400) {
      ...GatsbyImageSharpFixed
      // highlight-end
    }
  }
}
```

Read more about fixed image queries in the gatsby-plugin-sharp README.

Images that stretch across a *fluid* container

Create flexible sizes for an image that stretches to fill its container. E.g. for a container whose max width is 800px, the automatic sizes would be: 200px, 400px, 800px, 1200px and 1600px – enough to provide close to the optimal image size for every device size / screen resolution. If you want more control over which sizes are output you can use the `srcSetBreakpoints` parameter.

Once you've queried for a `fluid` image to retrieve its data, you can pass that data into the `Img` component:

```
import { useStaticQuery, graphql } from "gatsby"
import Img from "gatsby-image"

export default function Image() {
  const data = useStaticQuery(graphql`
    query {
      file(relativePath: { eq: "images/default.jpg" }) {
        childImageSharp {
          # Specify a fluid image and fragment
          # The default maxWidth is 800 pixels
          // highlight-start
          fluid {
            ...GatsbyImageSharpFluid
          }
          // highlight-end
        }
      }
    }
  `)
  return (
    <div>
      <h1>Hello gatsby-image</h1>
      <Img
        fluid={data.file.childImageSharp.fluid} {/* highlight-line */}
        alt="Gatsby Docs are awesome"
      />
    </div>
  )
}
```

Fluid image query parameters In a query, you can specify options for fluid images.

- `maxWidth` (int, default: 800)
- `maxHeight`(int)
- `quality` (int, default: 50)

- `srcSetBreakpoints` (array of int, default: [])
- `background` (string, default: `rgba(0,0,0,1)`)

Returns

- `base64` (string)
- `aspectRatio` (float)
- `src` (string)
- `srcSet` (string)
- `srcSetType` (string)
- `sizes` (string)
- `originalImg` (string)

This is where fragments like `GatsbyImageSharpFluid` come in handy, as they'll return all the above items in one line without having to type them all out:

```
file(relativePath: { eq: "images/default.jpg" }) {
  childImageSharp {
    // highlight-start
    fluid(maxWidth: 400) {
      ...GatsbyImageSharpFluid
      // highlight-end
    }
  }
}
```

Read more about fluid image queries in the gatsby-plugin-sharp README.

Resized images

In addition to *fixed* and *fluid* images, the gatsby-image API allows you to call a `resize` method with `gatsby-plugin-sharp` to return a single image as opposed to multiple sizes. There are no default fragments available for the resize method.

Parameters

- `width` (int, default: 400)
- `height` (int)
- `quality` (int, default: 50)
- `jpegProgressive` (bool, default: true)
- `pngCompressionLevel` (int, default: 9)
- `base64`(bool, default: false)

Returns Resize returns an object with the following items:

- `src` (string)
- `width` (int)
- `height` (int)

- `aspectRatio` (float)

```
allImageSharp {
  edges {
    node {
        resize(width: 150, height: 150, grayscale: true) {
          src
        }
    }
  }
}
```

Read more about resized image queries in the gatsby-plugin-sharp README.

Shared query parameters

In addition to `gatsby-plugin-sharp` settings in `gatsby-config.js`, there are additional query options that apply to *fluid*, *fixed*, and *resized* images:

- `grayscale` (bool, default: false)
- `duotone` (bool|obj, default: false)
- `toFormat` (string, default: “)
- `cropFocus` (string, default: `ATTENTION`)
- `fit` (string, default: `COVER`)
- `pngCompressionSpeed` (int, default: 4)
- `rotate` (int, default: 0)

Here's an example of using the `duotone` option with a fixed image:

```
fixed(
  width: 800,
  duotone: {
    highlight: "#f00e2e",
    shadow: "#192550"
  }
)
```

Duotone | Before - After

And an example of using the `grayscale` option with a fixed image:

```
fixed(
  grayscale: true
)
```

Grayscale | Before - After

Read more about shared image query parameters in the `gatsby-plugin-sharp` README.

Image query fragments

GraphQL includes a concept called "query fragments", which are a part of a query that can be reused. To ease building with `gatsby-image`, Gatsby image processing plugins which support `gatsby-image` ship with fragments which you can easily include in your queries.

> Note: using fragments in your queries depends on which data source(s) you have configured. Read more about image query fragments in the gatsby-image README.

Common fragments with `gatsby-transformer-sharp`

Fixed images

- `GatsbyImageSharpFixed`
- `GatsbyImageSharpFixed_noBase64`
- `GatsbyImageSharpFixed_tracedSVG`
- `GatsbyImageSharpFixed_withWebp`
- `GatsbyImageSharpFixed_withWebp_noBase64`
- `GatsbyImageSharpFixed_withWebp_tracedSVG`

Fluid images

- `GatsbyImageSharpFluid`
- `GatsbyImageSharpFluid_noBase64`
- `GatsbyImageSharpFluid_tracedSVG`
- `GatsbyImageSharpFluid_withWebp`
- `GatsbyImageSharpFluid_withWebp_noBase64`
- `GatsbyImageSharpFluid_withWebp_tracedSVG`

About `noBase64` If you don't want to use the blur-up effect, choose the fragment with `noBase64` at the end.

About `tracedSVG` If you want to use the traced placeholder SVGs, choose the fragment with `tracedSVG` at the end.

About `withWebp` If you want to automatically use WebP images when the browser supports the file format, use the `withWebp` fragments. If the browser doesn't support WebP, `gatsby-image` will fall back to the default image format.

Here's an example of using a non-default fragment from `gatsby-transformer-sharp`. Be sure to pick one that matches your desired image type (*fixed* or *fluid*):

```
file(relativePath: { eq: "images/default.jpg" }) {
  childImageSharp {
    fluid {
      // highlight-next-line
```

```
      ...GatsbyImageSharpFluid_tracedSVG
    }
  }
}
```

For more info on how these options work, check out the Gatsby Image demo: https://using-gatsby-image.gatsbyjs.org/

Additional plugin fragments Additionally, plugins supporting `gatsby-image` currently include gatsby-source-contentful, gatsby-source-datocms and gatsby-source-sanity. See the gatsby-image README for more details.

Gatsby-image props

After you've made a query, you can pass additional options to the gatsby-image component.

Name	Type	Description
`fixed`	`object`	Data returned from the `fixed` query
`fluid`	`object`	Data returned from the `fluid` query
`fadeIn`	`bool`	Defaults to fading in the image on load
`durationFadeIn`	`number`	fade-in duration is set up to 500ms by default
`title`	`string`	Passed to the rendered HTML `img` element
`alt`	`string`	Passed to the rendered HTML `img` element. Defaults to an empty string, e.g. `alt=""`
`crossOrigin`	`string`	Passed to the rendered HTML `img` element
`className`	`string` / `object`	Passed to the wrapper element. Object is needed to support Glamor's css prop
`style`	`object`	Spread into the default styles of the wrapper element
`imgStyle`	`object`	Spread into the default styles of the actual `img` element
`placeholderStyle`	`object`	Spread into the default styles of the placeholder `img` element
`placeholderClassName`	`string`	A class that is passed to the placeholder `img` element
`backgroundColor`	`string` / `bool`	Set a colored background placeholder. If true, uses "lightgray" for the color. You can also pass in any valid color string.
`onLoad`	`func`	A callback that is called when the full-size image has loaded.
`onStartLoad`	`func`	A callback that is called when the full-size image starts loading, it gets the parameter `{ wasCached: <boolean> }` provided.
`onError`	`func`	A callback that is called when the image fails to load.

Name	Type	Description
`Tag`	`string`	Which HTML tag to use for wrapping elements. Defaults to `div`.
`objectFit`	`string`	Passed to the `object-fit-images` polyfill when importing from `gatsby-image/withIEPolyfill`. Defaults to `cover`.
`objectPosition`	`string`	Passed to the `object-fit-images` polyfill when importing from `gatsby-image/withIEPolyfill`. Defaults to `50% 50%`.
`loading`	`string`	Set the browser's native lazy loading attribute. One of `lazy`, `eager` or `auto`. Defaults to `lazy`.
`critical`	`bool`	Opt-out of lazy-loading behavior. Defaults to `false`. Deprecated, use `loading` instead.

Here are some usage examples:

```
<Img
  fluid={data.file.childImageSharp.fluid}
  alt="Cat taking up an entire chair"
  fadeIn={false}
  className="customImg"
  placeholderStyle={{ `backgroundColor`: `black` }}
  onLoad={() => {
    // do loading stuff
  }}
  onStartLoad={({ wasCached }) => {
    // do stuff on start of loading
    // optionally with the wasCached boolean parameter
  }}
  onError={(error) => {
    // do error stuff
  }}
  Tag="custom-image"
  loading="eager"
/>
```