gatsby-image

Speedy, optimized images without the work.

`gatsby-image` is a React component specially designed to work seamlessly with Gatsby's GraphQL queries. It combines Gatsby's native image processing capabilities with advanced image loading techniques to easily and completely optimize image loading for your sites. `gatsby-image` uses gatsby-plugin-sharp to power its image transformations.

*Note: gatsby-image is **not** a drop-in replacement for `<img />`. It's optimized for fixed width/height images and images that stretch the full-width of a container. Some ways you can use `<img />` won't work with gatsby-image.*

Demo

Problem

Large, unoptimized images dramatically slow down your site.

But creating optimized images for websites has long been a thorny problem. Ideally you would:

- Resize large images to the size needed by your design
- Generate multiple smaller images so smartphones and tablets don't download desktop-sized images
- Strip all unnecessary metadata and optimize JPEG and PNG compression
- Efficiently lazy load images to speed initial page load and save bandwidth
- Use the "blur-up" technique or a "traced placeholder" SVG to show a preview of the image while it loads
- Hold the image position so your page doesn't jump while images load

Doing this consistently across a site feels like sisyphean labor. You manually optimize your images and then... several images are swapped in at the last minute or a design-tweak shaves 100px of width off your images.

Most solutions involve a lot of manual labor and bookkeeping to ensure every image is optimized.

This isn't ideal. Optimized images should be easy and the default.

Solution

With Gatsby, we can make images way *way* better.

`gatsby-image` is designed to work seamlessly with Gatsby's native image processing capabilities powered by GraphQL and Sharp. To produce perfect images, you need only:

1. Import `gatsby-image` and use it in place of the built-in `img`
2. Write a GraphQL query using one of the included GraphQL "fragments" which specify the fields needed by `gatsby-image`.

The GraphQL query creates multiple thumbnails with optimized JPEG and PNG compression. The `gatsby-image` component automatically sets up the "blur-up" effect as well as lazy loading of images further down the screen.

Install

`npm install --save gatsby-image`

Depending on the gatsby starter you used, you may need to include gatsby-transformer-sharp and gatsby-plugin-sharp as well, and make sure they are installed and included in your gatsby-config.

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

Then in your `gatsby-config.js`:

```
plugins: [`gatsby-transformer-sharp`, `gatsby-plugin-sharp`]
```

Also, make sure you have set up a source plugin, so your images are available in `graphql` queries. For example, if your images live in a project folder on the local filesystem, you would set up `gatsby-source-filesystem` in `gatsby-config.js` like so:

```
const path = require(`path`)

module.exports = {
  plugins: [
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        name: `images`,
        path: path.join(__dirname, `src`, `images`),
      },
    },
    `gatsby-plugin-sharp`,
    `gatsby-transformer-sharp`,
  ],
}
```

How to use

This is what a component using `gatsby-image` looks like:

```
import React from "react"
import { graphql } from "gatsby"
import Img from "gatsby-image"

export default ({ data }) => (
  <div>
    <h1>Hello gatsby-image</h1>
    <Img fixed={data.file.childImageSharp.fixed} />
  </div>
)

export const query = graphql`
  query {
    file(relativePath: { eq: "blog/avatars/kyle-mathews.jpeg" }) {
      childImageSharp {
        # Specify the image processing specifications right in the query.
        # Makes it trivial to update as your page's design changes.
        fixed(width: 125, height: 125) {
          ...GatsbyImageSharpFixed
        }
      }
    }
  }
`
```

For other explanations of how to get started with gatsby-image, see this blog post by community member Kyle Gill Image Optimization Made Easy with Gatsby.js or this one by Hunter Chang (which also includes some details about changes to gatsby-image for Gatsby v2): An Intro To Gatsby Image V2

Polyfilling object-fit/object-position for IE

If you'd like to include a polyfill for the `object-fit` / `object-position` CSS properties (which aren't supported by default in Internet Explorer), import from `gatsby-image/withIEPolyfill` instead:

```
// Other imports...
import Img from "gatsby-image/withIEPolyfill"

export default ({ data }) => (
  <div>
    <h1>Hello gatsby-image</h1>
    <Img
      fixed={data.file.childImageSharp.fixed}
      objectFit="cover"
      objectPosition="50% 50%"
      alt=""
    />
  </div>
)

// GraphQL query...
```

Importing from `gatsby-image/withIEPolyfill` tells Gatsby to automatically apply the `object-fit-images` polyfill to your image. To make your `object-fit` / `object-position` values work in IE, be sure to use the `objectFit` and `objectPosition` props (rather than the `imgStyle` prop or a CSS or CSS-in-JS solution) so the polyfill will recognize them.

Two types of responsive images

There are two types of responsive images supported by gatsby-image.

1. Images that have a *fixed* width and height

2. Images that stretch across a *fluid* container

In the first scenario, you want to vary the image's size for different screen resolutions -- in other words, create retina images.

For the second scenario, you want to create multiple sizes of thumbnails for devices with widths stretching from smartphone to wide desktop monitors.

To decide between the two, ask yourself: "do I know the exact size this image will be?" If yes, it's the first type. If no and its width and/or height need to vary depending on the size of the screen, then it's the second type.

In Gatsby's GraphQL implementation, you query for the first type by querying a child object of an image called `fixed` — which you can see in the sample component above. For the second type, you do a similar query but for a child object called `fluid`.

Fragments

GraphQL includes a concept called "query fragments". Which, as the name suggests, are a part of a query that can be used in multiple queries. 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, due to a limitation of GraphiQL, you can not currently use these fragments in the GraphiQL IDE.

Plugins supporting `gatsby-image` currently include gatsby-transformer-sharp, gatsby-source-contentful, gatsby-source-datocms and gatsby-source-sanity.

Their fragments are:

gatsby-transformer-sharp

- `GatsbyImageSharpFixed`
- `GatsbyImageSharpFixed_noBase64`

- `GatsbyImageSharpFixed_tracedSVG`
- `GatsbyImageSharpFixed_withWebp`
- `GatsbyImageSharpFixed_withWebp_noBase64`
- `GatsbyImageSharpFixed_withWebp_tracedSVG`
- `GatsbyImageSharpFluid`
- `GatsbyImageSharpFluid_noBase64`
- `GatsbyImageSharpFluid_tracedSVG`
- `GatsbyImageSharpFluid_withWebp`
- `GatsbyImageSharpFluid_withWebp_noBase64`
- `GatsbyImageSharpFluid_withWebp_tracedSVG`

gatsby-source-contentful

- `GatsbyContentfulFixed`
- `GatsbyContentfulFixed_noBase64`
- `GatsbyContentfulFixed_tracedSVG`
- `GatsbyContentfulFixed_withWebp`
- `GatsbyContentfulFixed_withWebp_noBase64`
- `GatsbyContentfulFluid`
- `GatsbyContentfulFluid_noBase64`
- `GatsbyContentfulFluid_tracedSVG`
- `GatsbyContentfulFluid_withWebp`
- `GatsbyContentfulFluid_withWebp_noBase64`

gatsby-source-datocms

- `GatsbyDatoCmsFixed`

- `GatsbyDatoCmsFixed_noBase64`
- `GatsbyDatoCmsFluid`
- `GatsbyDatoCmsFluid_noBase64`

gatsby-source-sanity

- `GatsbySanityImageFixed`
- `GatsbySanityImageFixed_noBase64`
- `GatsbySanityImageFluid`
- `GatsbySanityImageFluid_noBase64`

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

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.

Please see the gatsby-plugin-sharp documentation for more information on `tracedSVG` and its configuration options.

"Fixed" queries

Component

Pass in the data returned from the `fixed` object in your query via the `fixed` prop. e.g. `<Img fixed={fixed} />`

Query

```
{
  imageSharp {
    # Other options include height (set both width and height to crop),
    # grayscale, duotone, rotate, etc.
    fixed(width: 400) {
      # Choose either the fragment including a small base64ed image, a traced placeholder SVG, or one withou
      ...GatsbyImageSharpFixed
    }
  }
}
```

"Fluid" queries

Component

Pass in the data returned from the `fluid` object in your query via the `fluid` prop. e.g. `<Img fluid={fluid} />`

Query

```
{
  imageSharp {
    # i.e. the max width of your container is 700 pixels.
    #
    # Other options include maxHeight (set both maxWidth and maxHeight to crop),
    # grayscale, duotone, rotate, etc.
    fluid(maxWidth: 700) {
      # Choose either the fragment including a small base64ed image, a traced placeholder SVG, or one withou
      ...GatsbyImageSharpFluid_noBase64
    }
  }
}
```

Avoiding stretched images using the fluid type

As mentioned previously, images using the *fluid* type are stretched to match the container's width. In the case where the image's width is smaller than the available viewport, the image will stretch to match the container, potentially leading to unwanted problems and worsened image quality.

To counter this edge case one could wrap the *Img* component in order to set a better, for that case, `maxWidth`:

```
const NonStretchedImage = props => {
  let normalizedProps = props
  if (props.fluid && props.fluid.presentationWidth) {
    normalizedProps = {
      ...props,
      style: {
        ...(props.style || {}),
        maxWidth: props.fluid.presentationWidth,
        margin: "0 auto", // Used to center the image
      },
    }
  }

  return <Img {...normalizedProps} />
}
```

Note: The `GatsbyImageSharpFluid` fragment does not include `presentationWidth`. You will need to add it in your graphql query as is shown in the following snippet:

```
{
  childImageSharp {
    fluid(maxWidth: 500, quality: 100) {
      ...GatsbyImageSharpFluid
      presentationWidth
    }
    }
  }
```

Art-directing multiple images

`gatsby-image` supports showing different images at different breakpoints, which is known as art direction. To do this, you can define your own array of `fixed` or `fluid` images, along with a `media` key per image, and pass it to `gatsby-image`'s `fixed` or `fluid` props. The `media` key that is set on an image can be any valid CSS media query.

```
import React from "react"
import { graphql } from "gatsby"
import Img from "gatsby-image"

export default ({ data }) => {
  // Set up the array of image data and `media` keys.
  // You can have as many entries as you'd like.
  const sources = [
    data.mobileImage.childImageSharp.fluid,
    {
      ...data.desktopImage.childImageSharp.fluid,
      media: `(min-width: 768px)`,
    },
  ]

  return (
    <div>
      <h1>Hello art-directed gatsby-image</h1>
      <Img fluid={sources} />
    </div>
  )
}

export const query = graphql`
  query {
    mobileImage(relativePath: { eq: "blog/avatars/kyle-mathews.jpeg" }) {
      childImageSharp {
        fluid(maxWidth: 1000, quality: 100) {
          ...GatsbyImageSharpFluid
        }
      }
    }
    desktopImage(
      relativePath: { eq: "blog/avatars/kyle-mathews-desktop.jpeg" }
    ) {
      childImageSharp {
        fluid(maxWidth: 2000, quality: 100) {
          ...GatsbyImageSharpFluid
        }
      }
    }
  }
`
```

While you could achieve a similar effect with plain CSS media queries, `gatsby-image` accomplishes this using the `<picture>` tag, which ensures that browsers only download the image they need for a given breakpoint.

`gatsby-image` props

Name	Type	Description
`fixed`	`object` / `array`	Data returned from the `fixed` query. When prop is an array it has to be combined with `media` keys, allows for art directing `fixed` images.
`fluid`	`object` / `array`	Data returned from the `fluid` query. When prop is an array it has to be combined with `media` keys, allows for art directing `fluid` images.
`fadeIn`	`bool`	Defaults to fading in the image on load

Name	Type	Description
`durationFadeIn`	`number`	fading duration is set up to 500ms by default
`title`	`string`	Passed to the `img` element
`alt`	`string`	Passed to the `img` element. Defaults to an empty string `alt=""`
`crossOrigin`	`string`	Passed to the `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.
`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`.

Name	Type	Description
`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.
`fixedImages`	`array`	An array of objects returned from `fixed` queries. When combined with `media` keys, allows for art directing `fixed` images.
`fluidImages`	`array`	An array of objects returned from `fluid` queries. When combined with `media` keys, allows for art directing `fluid` images.
`draggable`	`bool`	Set the img tag draggable to either `false`, `true`

Image processing arguments

gatsby-plugin-sharp supports many additional arguments for transforming your images like `quality`, `sizeByPixelDensity`, `pngCompressionLevel`, `cropFocus`, `greyscale` and many more. See its documentation for more.

Some other stuff to be aware of

- If you want to set `display: none;` on a component using a `fixed` prop, you need to also pass in to the style prop `{ display: 'inherit' }`.
- Be aware that from a SEO perspective it is advisable not to change the image parameters lightheartedly once the website has been published. Every time you change properties within *fluid* or *fixed* (like *quality* or

maxWidth), the absolute path of the image changes. These properties generate the hash we use in our absolute path. This happens even if the image didn't change its name. As a result, the image could appear on the image SERP as "new" one. (more details can be found on this issue)

- By default, images don't load until JavaScript is loaded. Gatsby's automatic code splitting generally makes this fine but if images seem slow coming in on a page, check how much JavaScript is being loaded there.
- Images marked as `critical` will start loading immediately as the DOM is parsed, but unless `fadeIn` is set to `false`, the transition from placeholder to final image will not occur until after the component is mounted.
- gatsby-image is now backed by the newer `<picture>` tag. This newer standard allows for media types to be chosen by the browser without using JavaScript. It also is backward compatible to older browsers (IE 11, etc)
- Gifs can't be resized the same way as pngs and jpegs, unfortunately—if you try to use a gif with `gatsby-image`, it won't work. For now, the best workaround is to import the gif directly.
- Lazy loading behavior is dependent on `IntersectionObserver` which is not available in some fairly common browsers including Safari and IE. A polyfill is recommended.