This guide will show you how to configure your images, including choosing layouts, placeholders and image processing options. While most of these options are available regardless of where you source your images, be sure to refer to the documentation of your source plugin if you are using images from a CMS, as the exact options are likely to vary.

Components

The Gatsby Image plugin includes two components to display responsive images on your site. One is used for static and the other for dynamic images.

- **`StaticImage`** : Use this if the image is the same every time the component is used. *Examples: site logo, index page hero image*
- **`GatsbyImage`** : Use this if the image is passed into the component as a prop, or otherwise changes. *Examples: Blog post hero image, author avatar*

If you would like to learn how to set up the image plugins and use these components on your site, see the how-to guide.

These are props that can be passed to the components:

Shared props

The following props can be passed to both `GatsbyImage` and `StaticImage`. You may also use any valid `<img>` tag props, which are forwarded to the underlying `<img>` element.

Prop	Type	Default	Description
`alt` (Required)	`string`		Alternative text, passed to the `<img>` tag. Required for accessibility.
`as`	`ElementType`	`"div"`	The HTML element used for the outer wrapper.
`loading`	`"eager" | "lazy"`	`"lazy"`	Loading behavior for the image. You should set this to `"eager"` for above-the-fold images to ensure they start loading before React hydration.
`className`	`string`		CSS class applied to the outer wrapper.
`imgClassName`	`string`		CSS class applied to the `<img>` element.
`style`	`CSSProperties`		Inline styles applied to the outer wrapper.
`imgStyle`	`CSSProperties`		Inline styles applied to the `<img>` element.
`backgroundColor`	`string`	`transparent`	Background color applied to the wrapper.
`objectFit`	See MDN doc	`cover`	Resizing behavior for the image within its container.
`objectPosition`	See MDN doc	`50% 50%`	Position of the image within its container.

`StaticImage`

The `StaticImage` component can take all image options as props, as well as all shared props. Additionally, it takes this prop:

Prop	Type	Description
`src` (Required)	`string`	Source image, processed at build time. Can be a path relative to the source file, or an absolute URL.

Restrictions on using `StaticImage`

The images are loaded and processed at build time, so there are restrictions on how you pass props to the component. The values need to be statically-analyzed at build time, which means you can't pass them as props from outside the component, or use the results of function calls, for example. You can either use static values, or variables within the component's local scope. See the following examples:

This does not work:

```
// ⚠ Doesn't work

export function Logo({ logo }) {
  // You can't use a prop passed into the parent component
  return <StaticImage src={logo} />
}
```

...and nor does this:

```
// ⚠ Doesn't work

export function Dino() {
  // Props can't come from function calls
  const width = getTheWidthFromSomewhere()
  return <StaticImage src="trex.png" width={width} />
}
```

You can use variables and expressions if they're in the scope of the file, e.g.:

```
// OK
export function Dino() {
  // Local variables are fine
  const width = 300
  return <StaticImage src="trex.png" width={width} />
}
```

```
// Also OK

// A variable in the same file is fine.
const width = 300

export function Dino() {
  // This works because the value can be statically-analyzed
  const height = (width * 16) / 9
  return <StaticImage src="trex.png" width={width} height={height} />
}
```

If you find yourself wishing you could use a prop for the image `src` then it's likely that you should be using a dynamic image.

Using `StaticImage` with CSS-in-JS libraries

The `StaticImage` component does not support higher-order components, which includes the `styled` function from libraries such as Emotion and styled-components. The parser relies on being able to identify `StaticImage` components in the source, and passing them to a function means this is not possible.

```
// ⚠ Doesn't work

const AwesomeImage = styled(StaticImage)`
  border: 4px green dashed;
`

export function Dino() {
  // Parser doesn't know that this is a StaticImage
  return <AwesomeImage src="trex.png" />
}
```

If you use Emotion you can use the provided `css` prop instead:

```
// Emotion

export function Dino() {
  return (
    <StaticImage
      src="trex.png"
      css={css`
        border: 4px green dashed;
      `}
    />
  )
}
```

Unfortunately the `css` prop from styled-components turns the code into a `styled` function under the hood and as explained above `StaticImage` doesn't support that syntax.

You can also use a regular `style` or `className` prop. Note that in all of these cases the styling is applied to the wrapper, not the image itself. If you need to style the `<img>` tag, you can use `imgStyle` or `imgClassName`. The `className` or `imgClassName` prop is helpful if your styling library is giving you a computed class name string instead of the computed styles (e.g. if you use libraries like linaria).

`GatsbyImage`

This component accepts all shared props, as well as the one below. These props are passed directly to the component, and are not to be confused with image options, which are passed to the GraphQL resolver when using dynamic images.

Prop	Type	Description

`image` (Required)	`GatsbyImageData`	The image data object, returned from the `gatsbyImageData` resolver.

Image options

There are a few differences between how you specify options for `StaticImage` and `GatsbyImage`:

1. **How to pass options:** When using `StaticImage`, options are passed as props to the component, whereas for the `GatsbyImage` component they are passed to the `gatsbyImageData` GraphQL resolver.
2. **Option values:** In the `StaticImage` component, props such as `layout` and `placeholder` take a *string*, while the resolver takes a *a GraphQL enum*, which is upper case by convention and is not quoted like a string. Both syntaxes are shown in the reference below.

Important: For dynamic images, these options are for the `gatsbyImageData` resolver on sharp nodes. If you are using `gatsbyImageData` from a different plugin, such as a CMS or image host, you should refer to that plugin's documentation for the options, as they will differ from these. Static images use sharp under the hood, so these options apply when using the `StaticImage` component too.

It is a very good idea to use the GraphiQL IDE when writing your `gatsbyImageData` queries. It includes auto-complete and inline documentation for all of the options and lets you see the generated image data right inside the browser.

Both static and dynamic images have the following options available:

Option	Default string/enum value	Description
`layout`	`"constrained"` / `CONSTRAINED`	Determines the size of the image and its resizing behavior.
`aspectRatio`	Source image aspect ratio	Force a specific ratio between the image's width and height.
`width`/`height`	Source image size	Change the size of the image.
`placeholder`	`"dominantColor"`/`DOMINANT_COLOR`	Choose the style of temporary image shown while the full image loads.
`formats`	`["auto","webp"]`/`[AUTO,WEBP]`	File formats of the images generated.
`transformOptions`	`{fit: "cover", cropFocus: "attention"}`/`{fit: COVER, cropFocus: ATTENTION}`	Options to pass to sharp to control cropping and other image manipulations.

See all options.

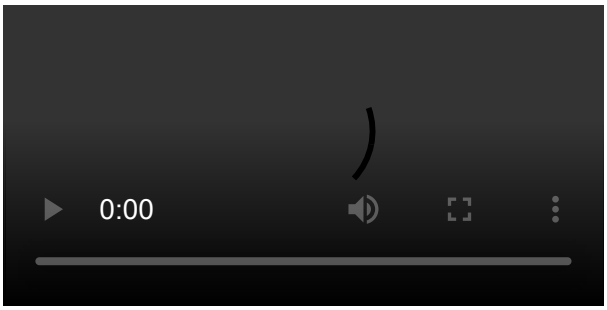
`layout`

The image components support three types of layout, which determine the image sizes that are generated, as well as the resizing behavior of the image itself in the browser.

Layout	Component prop value	Resolver prop value	Description

Constrained	`"constrained"`	`CONSTRAINED`	This is the default layout. It displays the image at the size of the source image, or you can set a maximum size by passing in `width` or `height`). If the screen or container size is less than the width of the image, it scales down to fit, maintaining its aspect ratio. It generates smaller versions of the image so that a mobile browser doesn't need to load the full-size image.
Fixed	`"fixed"`	`FIXED`	This is a fixed-size image. It will always display at the same size, and will not shrink to fit its container. This is either the size of the source image, or the size set by the `width` and `height` props. Only use this if you are certain that the container will never need to be narrower than the image.
Full width	`"fullWidth"`	`FULL_WIDTH`	Use this for images that are always displayed at the full width of the screen, such as banners or hero images. Like the constrained layout, this resizes to fit the container. However it is not restricted to a maximum size, so will grow to fill the container however large it is, maintaining its aspect ratio. It generates several smaller image sizes for different screen breakpoints, so that the browser only needs to load one large enough to fit the screen. You can pass a `breakpoints` prop if you want to specify the sizes to use, though in most cases you can allow it to use the default.

You can compare the layouts in the video below:



To set the layout, pass in the type to the layout prop. e.g.

```
<StaticImage
  src="./dino.png"
  alt="A dinosaur"
  // highlight-next-line
  layout="fixed"
/>
```

For a dynamic image, pass it to the resolver:

```
dino {
  childImageSharp {
    # highlight-next-line
```

```
    gatsbyImageData(layout: FIXED)
  }
}
```

`width` / `height`

Layouts: fixed and constrained

Size props are optional in `GatsbyImage` and `StaticImage`. Because the images are processed at build time, the plugin knows the size of the source image and can add the correct width and height to the `<img>` tag, so it displays correctly with no layout jumping. However, if you want to change the display size you can use the size options to do this.

For a `fixed` layout, these define the size of the image displayed on screen. For a `constrained` image these define the maximum size, as the image will scale down to fit smaller containers if needed.

If you set just one of these, the source image is resized to that width or height while maintaining aspect ratio. If you include both then it is also cropped if needed to ensure it is that exact size.

`aspectRatio`

Layouts: fixed, constrained, and fullWidth

The `aspectRatio` prop forces an image to the specified aspect ratio, cropping if needed. The value is a number, but can be clearer to express as a fraction, e.g. `aspectRatio={16/9}`

For `fixed` and `constrained` images, you can also optionally pass either a `width` or `height`, and it will use that to calculate the other dimension. For example, if you pass `width={800} aspectRatio={4/3}` then `height` will be set to the width divided by the aspect ratio: so `600`. Passing `1` as the aspectRatio will crop the image to a square. If you don't pass a width or height then it will use the source image's width.

For `fullWidth` images you don't specify width or height, as it resizes to fit the screen width. Passing `aspectRatio` will crop the image if needed, and the height will scale according to the width of the screen. For example, if you set the aspectRatio to `16/9` then when the image is displayed full width on a screen that is 1280px wide, the image will be 720 pixels high.

Note: There are several advanced options that you can pass to control the cropping and resizing behavior. For more details, see the `transformOptions` reference.

`placeholder`

Gatsby image components are lazy-loaded by default, which means that if they are offscreen they are not loaded by the browser until they come into view. To ensure that the layout does not jump around, a placeholder is displayed before the image loads. You can choose one of three types of placeholder (or not use a placeholder at all):

Placeholder	Component prop value	Resolver prop value	Description
Dominant color	`"dominantColor"`	`DOMINANT_COLOR`	The default placeholder. This calculates the dominant color of the source image and uses it as a solid background color.
Blurred	`"blurred"`	`BLURRED`	This generates a very low-resolution version of the source image and displays it as a blurred

			background.
Traced SVG	`"tracedSVG"`	`TRACED_SVG`	This generates a simplified, flat SVG version of the source image, which it displays as a placeholder. This works well for images with simple shapes or that include transparency.
None	`"none"`	`NONE`	No placeholder. You can use the background color option to set a static background if you wish.

`formats`

Default component prop value: `["auto", "webp"]`. *Default resolver prop value:* `[AUTO, WEBP]`

The Gatsby Image plugin supports four output formats: JPEG, PNG, WebP and AVIF. By default, the plugin generates images in the same format as the source image, as well as WebP. For example, if your source image is a PNG, it will generate PNG and WebP images. In most cases, you should not change this. However, in some cases you may need to manually set the formats. One reason for doing so is if you want to enable support for AVIF images. AVIF is a new image format that results in significantly smaller file sizes than alternative formats. It currently has limited browser support, but this is likely to increase. It is safe to include as long as you also generate fallbacks for other browsers, which the image plugin does automatically by default.

`transformOptions`

These values are passed in as an object to `transformOptions`, either as a prop to `StaticImage`, or to the resolver for dynamic images. They are advanced settings that most people will not need to change. Any provided object is merged with the defaults below.

Option name	Default	Description
`grayscale`	`false`	Convert image to grayscale
`duotone`	`false`	Add duotone effect. Pass `false`, or options object containing `{highlight: string, shadow: string, opacity: number}`
`rotate`	`auto`	Rotate the image. Value in degrees.
`trim`	`10`	Trim "boring" pixels. Value is the threshold. See the sharp documentation.
`cropFocus`	`"attention"`/`ATTENTION`	Controls crop behavior. See the sharp documentation for strategy, position and gravity.
`fit`	`"cover"`/`COVER`	Controls behavior when resizing an image and proving both width and height. See the sharp documentation.

All options

The Gatsby Image plugin uses sharp for image processing, and supports passing through many advanced options, such as those affecting cropping behavior or image effects including grayscale or duotone, as well as options specific to each format.

Option	Default	Description

layout	"constrained" / CONSTRAINED	Determines the size of the image and its resizing behavior.
width/height	Source image size	Change the size of the image.
aspectRatio	Source image aspect ratio	Force a specific ratio between the image's width and height.
placeholder	"dominantColor"/DOMINANT_COLOR	Choose the style of temporary image shown while the full image loads.
formats	["auto","webp"]/[AUTO,WEBP]	File formats of the images generated.
transformOptions	{fit: "cover", cropFocus: "attention"}	Options to pass to sharp to control cropping and other image manipulations.
sizes	Generated automatically	The <img> sizes attribute, passed to the img tag. This describes the display size of the image, and does not affect generated images. You are only likely to need to change this if your are using full width images that do not span the full width of the screen.
quality	50	The default image quality generated. This is overridden by any format-specific options
outputPixelDensities	For fixed images: [1, 2] For constrained: [0.25, 0.5, 1, 2]	A list of image pixel densities to generate. It will never generate images larger than the source, and will always include a 1x image. The value is multiplied by the image width, to give the generated sizes. For example, a 400 px wide constrained image would generate 100, 200, 400 and 800 px wide images by default. Ignored for full width layout images, which use breakpoints instead.
breakpoints	[750, 1080, 1366, 1920]	Output widths to generate for full width images. Default is to generate widths for common device resolutions. It will never generate an image larger than the source image. The browser will automatically choose the most appropriate.
blurredOptions	None	Options for the low-resolution placeholder image. Ignored unless placeholder is blurred.
tracedSVGOptions	None	Options for traced placeholder SVGs. See potrace options. Ignored unless placeholder is traced SVG.
jpgOptions	None	Options to pass to sharp when generating JPG images.
pngOptions	None	Options to pass to sharp when generating PNG images.

`webpOptions`	None	Options to pass to sharp when generating WebP images.
`avifOptions`	None	Options to pass to sharp when generating AVIF images.

Customizing the default options

You might find yourself using the same options (like `placeholder`, `formats` etc.) with most of your `GatsbyImage` and `StaticImage` instances. You can customize the default options with `gatsby-plugin-sharp`.

The following configuration describes the options that can be customized along with their default values:

```
module.exports = {
  plugins: [
    {
      resolve: `gatsby-plugin-sharp`,
      options: {
        defaults: {
          formats: [`auto`, `webp`],
          placeholder: `dominantColor`,
          quality: 50,
          breakpoints: [750, 1080, 1366, 1920],
          backgroundColor: `transparent`,
          tracedSVGOptions: {},
          blurredOptions: {},
          jpgOptions: {},
          pngOptions: {},
          webpOptions: {},
          avifOptions: {},
        },
      },
    },
    `gatsby-transformer-sharp`,
    `gatsby-plugin-image`,
  ],
}
```

Helper functions

There are a number of utility functions to help you work with `gatsbyImageData` objects. We strongly recommend that you do not try to access the internals of these objects directly, as the format could change.

`getImage`

Safely get a `gatsbyImageData` object. It accepts several different sorts of objects, and is null-safe, returning `undefined` if the object passed, or any intermediate children are undefined.

If passed a `File` object, it will return `file?.childImageSharp?.gatsbyImageData`. If passed a node such as a `ContentfulAsset` that includes a `gatsbyImageData` field, it will return the `gatsbyImageData` object. If passed a `gatsbyImageData` object itself, it will return the same object.

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

const image = getImage(data.avatar)

// This is the same as:

const image = data?.avatar?.childImageSharp?.gatsbyImageData
```

`getSrc`

Get the default image `src` as a string. This will be the fallback, so usually jpg or png. Accepts the same types as `getImage`.

```
import { getSrc } from "gatsby-plugin-image"
//...
const src = getSrc(data.hero)

return <meta property="og:image" content={src} />
```

`getSrcSet`

Get the default image `srcset`. This will be the fallback, so usually jpg or png.

`withArtDirection`

By default, the plugin displays different image resolutions at different screen sizes, but it also supports art direction, which is where a visually-different image is displayed at different sizes. This could include displaying a simplified logo or a tighter crop on a profile picture when viewing on a small screen. To do this, you can use the `withArtDirection` function. You need both images available from GraphQL, and you should be able to write a media query for each size.

The first argument is the default image. This is displayed when no media queries match, but it is also used to set the layout, size, placeholder and most other options. You then pass an array of "art directed images" which are objects with `media` and `image` values.

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

export function MyImage({ data }) {
  const images = withArtDirection(getImage(data.largeImage), [
    {
      media: "(max-width: 1024px)",
      image: getImage(data.smallImage),
    },
  ])
```

```
  return <GatsbyImage image={images} />
}
```

When the screen is less than 1024px wide, it will display `smallImage`. Otherwise, it will display `largeImage`.

The aspect ratio is set by the default image, and doesn't automatically change with the different sources. The way to handle this is to use CSS media queries. For example, you could use this CSS to change the size of the container in small images:

```
@media screen and (max-width: 1024px) {
  .art-directed {
    width: 400px;
    height: 300px;
  }
}
```

You can then apply this using plain CSS, or the styling system of your choice. e.g.

```
import { GatsbyImage, getImage, withArtDirection } from "gatsby-plugin-image"
import "./style.css"

export function MyImage({ data }) {
  const images = withArtDirection(getImage(data.largeImage), [
    {
      media: "(max-width: 1024px)",
      image: getImage(data.smallImage),
    },
  ])

  return <GatsbyImage className="art-directed" image={images} />
}
```