This is a technical document for anyone who is interested in how images are implemented in Gatsby. This is not for learning how to use images in Gatsby, or even how to add image support to a plugin. This is for if you are working on Gatsby, or if you're curious about how it's implemented.

The image plugins

Image processing in Gatsby has a large number of parts, with `gatsby-plugin-image` serving as a core, but using several other plugins in order to perform its job. We have tried to simplify this for end-users by documenting most of the features as if they were all part of `gatsby-plugin-image`, rather than trying to document the borders of responsibility between `gatsby-plugin-image`, `gatsby-plugin-sharp`, `gatsby-transformer-sharp` and `gatsby-source-filesystem`. Similarly, `StaticImage` is presented to the user as a React component, and they only need to interact with it as one. However it is in fact the front end to an image processing pipeline that includes three Gatsby plugins, two Babel plugins and several hooks into the Gatsby build process. This document will show exactly what each plugin does, and describe how the main parts are implemented. These are the plugins used for image processing and display.

`gatsby-plugin-image`

This is the main plugin that users interact with. It includes two components for displaying images, as well as helper functions and a toolkit for plugin developers.

The `GatsbyImage` component

This is a React component, and is the actual component used to display all images. If somebody uses `StaticImage`, or an image component from a CMS provider, they all use `GatsbyImage` under the hood for the actual image display.

The `StaticImage` component

A lightweight wrapper around `GatsbyImage`, this component is detected during the build process and the props are extracted to enable images to be downloaded, and processed by `gatsby-plugin-sharp` without needing GraphQL.

Plugin toolkit

`gatsby-plugin-image` includes several functions that are used by third-party source plugins to enable them to generate image data objects. This includes helpers in `gatsby-plugin-image/graphql-utils`, which help plugin authors create `gatsbyImageData` resolvers, as well as the `getImageData` function used to generate image data at runtime by plugins with URL-based image resizing APIs. It does not perform any actual image processing (and doesn't require sharp), but does ensure that the correct object is generated with all of the correct image sizes. It includes the `getLowResolutionImageURL` function which helps when generating blurred placeholders.

Runtime helpers

The plugin exports a number of other helper functions designed to help end-users work with image data objects at runtime. These include the `getImage`, `getSrc` and `getSrcSet` utilities, as well as the `withArtDirection` helper.

`gatsby-plugin-sharp`

This includes the actual image processing functions from sharp and potrace. It includes the functions that generate the image data object, including calculating which srcset sizes to generate. It exports `generateImageData`, which is used by `gatsby-transformer-sharp` and `gatsby-plugin-image`. It takes a `File` node and the image processing arguments, calculates which images to generate, processes these images and returns an image data object suitable for passing to `GatsbyImage`. It also exports helper functions for third party plugins to use, such as `traceSVG`.

`gatsby-transformer-sharp`

This plugin attaches a `childImageSharp` node to all image `File` nodes. This includes the `gatsbyImageData` resolver, which uses `generateImageData` from `gatsby-plugin-sharp` to process images and return an image data object suitable for passing to `GatsbyImage`. The node also includes legacy `fixed` and `fluid` resolvers for the old `gatsby-image` component.

`gatsby-core-utils`

Third party plugin authors can use the `fetchRemoteFile` function to download files and make use of Gatsby's caching without needing to create a file node. This is used for low-resolution placeholder images.

`gatsby-source-filesystem`

The image plugin uses `createRemoteFileNode` when a `StaticImage` component has a remote URL as `src`. It does not currently use `fetchRemoteFile`, because `generateImageData` requires a `File` object.

Third-party plugins

Many source plugins now support `GatsbyImage`. They do this by generating image data objects that can be passed to the `GatsbyImage` component, or by providing their own components that wrap it. This data is either returned from a custom `gatsbyImageData` resolver on the plugin's nodes, or via a runtime helper that accepts data from elsewhere. An example of the latter would be the new Shopify plugin, which includes a `getShopifyImage` function that can be used to generate images from the Shopify search or cart APIs. These plugins all use the plugin toolkit from `gatsby-plugin-image` to ensure that the object they create are compatible. These do not require sharp, as the CMS or CDN provider handles the resizing. The API for each plugin's `gatsbyImageData` resolver will depend on the individual plugin, but authors are encouraged to provide an API similar to the one from `gatsby-transformer-sharp`.

How `GatsbyImage` works

`GatsbyImage` is a React component used to display performant, responsive images in Gatsby. It is used under the hood by all other compatible components such as `StaticImage` and components from CMS source plugins. It uses several performance tricks, and deserves most of the credit for improved performance scores when switching to the image plugin.

Anatomy of the component

The `GatsbyImage` component wraps several other components, which are all exported by the plugin. It was originally designed to allow users to compose their own custom image components, but we have not documented this, so it should currently be considered unsupported. It is something that could be looked at in the future, but until that point `GatsbyImage` and `StaticImage` should be considered the only public components.

Lazy hydration

Throughout the component there are different versions delivered for browser and server. The reason for this is that the component performs lazy hydration: the image is loaded as soon as the SSR HTML is loaded in the browser, including blur-up and lazy-loaded images. Hydration is usually the slowest part of a React app's page load. `GatsbyImage` avoids this by skipping React for all initial image loads, leading to faster LCP. The plugin uses `onRenderBody` in `gatsby-ssr` to inject inline script and CSS tags to enable this. The JS attaches a `load` event listener to the body, which hides the placeholder and shows the main image when any `GatsbyImage` has loaded. It uses `<noscript>` tags to ensure that images still work with scripts disabled.

Inside the `GatsbyImage` browser component, this means the load handling is skipped for any component that was rendered in SSR (which is indicated by adding a `data-gatsby-image-ssr` prop in the server component). However for any images that do not have this prop, this load handling happens in React. This will be the case for any image rendered after initial load, such as after page navigation or conditional rendering.

The browser component is a class component, which uses `shouldComponentUpdate` to ensure that it is never hydrated as part of the normal rendering process in the browser. Instead, the image hydrates or renders itself manually, using the `lazyHydrate` function, which is itself loaded asynchronously using `import()`. This takes the `ref` of the wrapper's HTML element, and uses `hydrate` or `render` from `react-dom` to render or hydrate the inner components as their own React tree. This ensures that the hydration of the images never blocks page loading, contributing to faster TTI.

Sizer

The `GatsbyImage` component supports three types of layout, which define the resizing behavior of the images. The component uses a `Sizer` component inside `layout-wrapper.tsx` to ensure that the wrapper is the correct size, even before any image has loaded. This avoids the page needing to re-layout, which looks bad for the user and is a serious problem for performance. For fixed layout components the size is just set via CSS. For full-width images, `Sizer` uses a `padding-top` hack to maintain aspect ratio as the image scales infinitely. For constrained layouts, `Sizer` uses an `<img>` tag with an inline, empty SVG `src` to make the browser use its native `<img>` resizing behaviour, even though the main image will not have loaded at this point.

Placeholder

`GatsbyImage` supports displaying a placeholder while the main image loads. There are two kinds of placeholder that are currently supported: flat colors and images. The type of placeholder is set via the image data object, and will either be a data URI for the image, or a CSS color value. The image will either be a base64-encoded low resolution raster image (called `BLURRED` when using sharp) or a URI-encoded SVG (called `TRACED_SVG`). The raster image will by default be 20px wide, and the same aspect ratio as the main image. This will be resized to fill the full container, giving a blurred effect. The SVG image is expected to be a single-color, simplified SVG generated using potrace. While these are the defaults produced by sharp, and also used by many third-party source plugins, we do not enforce this, and it can be any URI. We strongly encourage the use of inline data URIs, as any placeholder that needs to make a network request will defeat much of the purpose of using a placeholder. The actual placeholder element is a regular `<img>` tag, even for SVGs.

The alternative placeholder is a flat color. This is expected to be calculated from the dominant color of the source image. sharp supports performing this calculation, and some CMSs provide it in the image metadata. This color is applied as a background color to a placeholder `<div>` element.

When the main image is loaded, the placeholder is faded-out using a 250ms CSS opacity transition. We do not currently support disabling or changing this fade-out time, but it is a common request so could be a useful addition.

Main image

The main image component displays the actual image, as defined in the image data object. There is a lot of flexibility in how this is rendered, depending on which formats are provided.

In most cases, the object will include multiple sources in next-gen format such as WebP or AVIF, plus one fallback in JPEG or PNG format. In these cases, the component will render a `<picture>` tag, with multiple `<source>` elements and an `<img>` tag for the fallback. The `<source>` and `<img>` tags will always include a `srcset` prop, with multiple image resolutions according to the layout and size of the source and input images. We strongly recommend that users and source plugin authors allow the image plugin to generate these automatically. We use `w` (pixel width) units for the `srcset` rather than pixel density values, as this offers the browser the most flexibility in choosing the source to download. If there are not multiple sources provided, then an `<img>` tag will be rendered without an enclosing `<picture>` tag.

We pass through `media` props to the `<source>` elements, allowing art direction to be used. However we do not attempt to do any handling of changing the aspect ratio of the container in these cases, so the user must do this themselves using CSS.

How `StaticImage` works

The image plugin performs a number of tricks so that the `StaticImage` component appears to work like a regular React component, while being able to process images at build time.

The problem

In order to process images at build time, Gatsby needs to know which images will be needed and how they will need to be sized. It needs to do this in the scope of the build process so it has access to the sharp plugin (so it can't be done in e.g. the Babel plugin) and it can't be done by actually rendering the page, because that will miss images that are conditionally rendered and has similar issues with access to sharp and the Node APIs. The solution we have used is to use static analysis of the source files. This is similar to how static queries are extracted from source files.

Build process

Extract `StaticImage` props

The static analysis happens in `gatsby-plugin-image`, during the `preprocessSource` lifecycle. This runs immediately before query extraction. The plugin uses Babel to find references to `StaticImage` imported from `gatsby-plugin-image`. It then uses babel-jsx-utils to extract the value of the props. This calls `evaluate()` on the AST nodes, so is able to extract values and evaluate expressions in the local scope as well as inline literals. If any of these props fails to be evaluated, it generates a structured error that includes the location of the error and a link to the docs.

Process images

If all is well, all the props have been extracted. The only required prop is `src`: all the others can use default values. These props are then passed to `generateImageData` from `gatsby-plugin-sharp`. This handles the actual image processing, and returns an `IGatsbyImageData` object. If the `src` is a remote URL then the image is downloaded and cached locally. This image data object includes all of the data needed to display an image, including a list of sources, dimensions, layout, placeholder etc (though no actual image data except for data URIs of placeholders).

Write out image data

This object is then written to disk as a JSON file. The filename is generated using a hash of the props, and it is saved in `.cache/caches/gatsby-plugin-image`. If there have been any errors, an object with the error data is written instead. This is so the errors can be displayed in the browser console too.

Injecting the data

The rest of the build process then happens, and the next step where the image plugin is involved is the rendering stage. The image plugin adds a local Babel plugin `babel-plugin-parse-static-images` which once again finds the `StaticImage` components and extracts the props. However this time, the props are only used to calculate the JSON filename hash. The plugin then adds a new `__imageData` prop to the component, which is a `require()` of the image data JSON file. If there are errors either during the parsing, or passed through in the JSON file, these are inserted as an `__error` prop.

Runtime

At runtime, the `StaticImage` component behaves as a regular React component, but with the special `__imageData` prop injected by Babel. The component itself is a lightweight wrapper around `GatsbyImage`, and aside from error handling it just passes `__imageData` to the wrapped `GatsbyImage` component.