Adding Gatsby Image support to your plugin

The new Gatsby image plugin includes React components for displaying images, and these can be used with data from plugins. The plugin handles all of the hard parts of displaying responsive images that follow best practices for performance. In fact we are confident that it is the fastest way to render images in React, as it can handle blur-up and lazy-loading before React hydration. Support for these are available out of the box in `gatsby-transformer-sharp`, so if your plugin downloads images and processes them locally then your users can use the `gatsbyImageData` resolver. However, if your CDN can deliver images of multiple sizes with a URL-based API, then the plugin includes a toolkit to allow you to give your users the same great experience without needing to download the images locally. It also allows you to create components that display these images dynamically at runtime, without needing to add a GraphQL resolver.

Adding a `gatsbyImageData` GraphQL resolver

You can give your users the best experience by adding a `gatsbyImageData` resolver to your image nodes. This allows you to generate low-resolution or traced SVG images as inline data URIs to use as placeholders. You can also calculate the image's dominant color for an alternative placeholder. These are the same placeholders that are included with `gatsby-transformer-sharp`, and will give the best experience for your users. If you are able to deliver these directly from your CMS or other data source then this is ideal, but otherwise you can use helper functions included in `gatsby-plugin-sharp`.

There are three steps to add a basic `gatsbyImageData` resolver:

1. Create the `generateImageSource` function
2. Create the resolver function
3. Add the resolver

Create the `generateImageSource` function

The `generateImageSource` function is where you generate your image URLs. The image plugin calculates which sizes and formats are needed, according to the format, size and breakpoints requested by the user. For each of these, your function is passed the base URL, width, height and format (i.e. the image filetype), as well as any custom options that your plugin needs. You then return the generated URL. The returned object also includes width, height and format. This means you can return a different value from the one requested. For example, if the function requests an unsupported format or size, you can return a different one which will be used instead.

```
// In this example we use a custom `quality` option
const generateImageSource = (baseURL, width, height, format, fit, options) => {
  const src = `https://myexampleimagehost.com/${baseURL}?w=${width}&h=${height}&fmt=${format
  return { src, width, height, format }
}
```

Create the resolver function

You can then use the function created in the previous step to build your resolver function. It can be an async function, and it should return the value from `generateImageData`. An example resolver could look like this:

```
import { generateImageData, getLowResolutionImageURL } from "gatsby-plugin-image"


const resolveGatsbyImageData = async (image, options) => {
   // The `image` argument is the node to which you are attaching the resolver,
   // so the values will depend on your data type.
  const filename = image.src

  const sourceMetadata = {
    width: image.width,
    height: image.height,
    // In this example, the node has a value like "image/png", which needs
    // converting to a value such as "png". If this omitted, the function will
    // attempt to work it out from the file extension.
    format: image.mimeType.split("/")[1]
  }

  const imageDataArgs = {
    ...options,
    // Passing the plugin name allows for better error messages
    pluginName: `gatsby-source-example`,
    sourceMetadata,
    filename,
    placeholderURL,
    generateImageSource,
    options,
  }

  // Generating placeholders is optional, but recommended
  if(options.placeholder === "blurred") {
    // This function returns the URL for a 20px-wide image, to use as a blurred placeholder
    // You need to download the image and convert it to a base64-encoded data URI
    const lowResImage = getLowResolutionImageURL(imageDataArgs)

    // This would be your own function to download and generate a low-resolution placeholde
    imageDataArgs.placeholderURL =  await getBase64Image(lowResImage)
  }

  // You could also calculate dominant color, and pass that as `backgroundColor`
  // gatsby-plugin-sharp includes helpers that you can use to generate a tracedSVG or calcu
  // the dominant color of a local file, if you don't want to handle it in your plugin


  return generateImageData(imageDataArgs)
}
```

Add the resolver

You should register the resolver using the `createResolvers` API hook. `gatsby-plugin-image/graphql-utils` includes an optional utility function to help with this. It registers the resolver with all of the base arguments needed to create the image data, such as width, aspect ratio, layout and background color. These are defined with comprehensive descriptions that are visible when your users are building queries in GraphiQL.

You can pass additional arguments supported by your plugin, for example, image options such as quality. However, if you want complete control over the resolver args, then you will want to create it yourself from scratch. We recommend keeping the args similar to the default, as this is what users will be expecting, and it means you benefit from the plugin documentation. At a minimum, you should always expose `layout`, `width` and `height` as args.

The arguments:

- `resolverFunction`: the resolver function that you created in step 1. It receives the node and the arguments and should return the image data object.
- `additionalArgs`: an object defining additional args, in the same format used by Gatsby Type Builders

For example, to add a `gatsbyImageData` resolver onto a `ProductImage` node that you have previously defined:

```
// Note the different import
import { getGatsbyImageResolver } from "gatsby-plugin-image/graphql-utils"

export function createResolvers({ createResolvers }) {
  createResolvers({
    ProductImage: {
      // loadImageData is your custom resolver, defined in step 2
      gatsbyImageData: getGatsbyImageResolver(loadImageData, {
        quality: "Int",
      }),
    },
  })
}
```

Adding a custom image component

If you have a URL-based image API, you can create a custom image component that wraps `<GatsbyImage />` and displays images generated at runtime. If you have a source plugin, this can accept your native image object or it can take a base URL and generate the image based on that. This is a good solution for image CDNs that aren't handling their own CMS data, and can generate a transformed image from a source URL and dimensions.

There are three steps to create a custom image component:

1. Create your URL builder function
2. Create your image data function
3. Create your wrapper component (optional)

Create your URL builder function

This is similar to the `generateImageSource` approach described above. The difference is that it returns a URL string. This is an example for the same image host:

```
function urlBuilder({ baseUrl, width, height, format, options }) {
  return `https://myexampleimagehost.com/${baseURL}?w=${width}&h=${height}&fmt=${format}&q=$
}
```

If your host supports it, we recommend using auto-format to deliver next-generation image formats such as WebP or AVIF to supported browsers. In this case, ignore the `format` option.

Create your image data function

This is similar to the image resolver described above. However, because it executes in the browser you can't use node APIs. Because it needs to run before the image loads, it should be fast and synchronous. You will not be downloading and generating base64 placeholders, for example, or calculating dominant colors. If you have these values pre-calculated then you can pass these in and use them, but they need to be available in the props that you pass to the function at runtime. Any placeholder that is generated asynchronously would defeat the purpose: it prevents SSR, and is almost certainly slower than just downloading the main image.

The function should accept the props that will be passed into your component, and at a minimum it needs to take the props required by the `getImageData` helper function from `gatsby-plugin-image`. Here is an example for an image host:

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

export function getExampleImageData({ image, ...props }) {
  return getImageData({
    baseUrl: image.url,
    sourceWidth: image.width,
    sourceHeight: image.height,
    urlBuilder,
    pluginName: "gatsby-source-example",
    // If your host supports auto-format/content negotiation, pass this as the formats arra
    formats: ["auto"],
    ...props,
  })
}
```

You can export this function as a public API, and users can use the function to generate data to pass to `GatsbyImage`:

```
// This might come from an API at runtime
const image = {
  url: "kitten.jpg",
  width: 800,
  height: 600,
}
const imageData = getExampleImageData({ image, layout: "fixed", width: 400 })
return <GatsbyImage image={imageData} alt="Kitten" />
```

Create your wrapper component

This stage is optional: you may prefer to share the image data function and let your users pass the result to `<GatsbyImage>`, as shown above. However, the developer experience is better with a custom image component.

The component should accept the same props as your image data function, as well as all of the props for `<GatsbyImage>` which it can pass down to that component. Here's how you might type the props in TypeScript:

```
interface ImageComponentProps
  //This is the type for your image data function
  extends GetGatsbyImageDataProps,
    // We omit "image" because that's the prop that we generate,
    Omit<GatsbyImageProps, "image"> {
  // Any other props can go here
  myCustomProp?: string
}
```

Your component can accept just a URL if that's enough to identify the image, or you can pass a full object. This can be an object that you have passed in through your GraphQL API, or it can be data coming from outside of Gatsby, such as via search results or a shopping cart API. Unlike the GraphQL resolvers or the built-in `StaticImage` component, this can be dynamic data that changes at runtime.

For best results, you should pass in the dimensions of the source image, so make sure to include that data if it's available. With dimensions, the plugin can calculate aspect ratio and the maximum size image to request.

The component itself should wrap `<GatsbyImage>`, using your image data function to generate the object to pass to it.

```
import * as React from "react"
import { GatsbyImage } from "gatsby-plugin-image"
import { getExampleImageData } from "./my-image-data"

export function ExampleImage({
  // Destructure the props that you are passing to the image data function
  image,
  width,
  height,
  layout,
  backgroundColor,
  sizes,
  aspectRatio,
  options,
  // Use ...rest for the GatsbyImage props
  ...props
}) {
  const imageData = getExampleImageData({
    image,
    width,
    height,
    layout,
    backgroundColor,
    sizes,
    aspectRatio,
    options,
  })

  // Pass the image data and spread the rest of the props
  return <GatsbyImage image={imageData} {...props} />
}
```

The user could then use the component like this:

```
// This might come from an API at runtime
const image = {
  url: "baseurl.jpg",
  width: 800,
  height: 600,
}

return (
  <ExampleImage
    image={image}
    layout="fixed"
    width={400}
    backgroundColor="#660033"
    alt="My image"
  />
)
```

A different component for an image CDN might not expect the user to know the dimensions of the source image, and might want to allow them to just pass a base URL:

```
<ExampleImage
  image="https://example.com/nnnnn/bighero.jpg"
  loading="eager"
  layout="fullWidth"
  aspectRatio={16 / 9}
  alt=""
/>
```

Other considerations

You should add `gatsby-plugin-image` as a peer dependency to your plugin, and tell your users to install it in your docs. Don't add it as a direct dependency, as this could lead to multiple versions being installed. You can refer users to the `gatsby-plugin-image` docs for instructions on using the component. However, be sure to document the specifics of your own resolver: it may not be immediately clear to users that the args are different from those in sharp. You may want to highlight specific differences, such as if you don't support AVIF images, or if you do support GIFs, as well as explaining the placeholders that you support.