Site Generators, frameworks, and CMSs

- Static Site Generators
- Frameworks
- Content Management Systems

While certainly an improvement over manually saving alternate cuts of each image and hand-optimizing them through a tool like Squoosh.app, automating image compression as a step in the development process has some limitations. For one, you may not always have full control over the images used throughout a site—most user-facing images on the web are *content* concerns more than development concerns, uploaded by users or editors, rather than living in a repository alongside development assets like JavaScript and stylesheets.

This will typically necessitate more than one process for image management: a development-level task for the image assets used in building and maintaining a site—backgrounds, icons, logos, and so on—and another concerned with image assets generated through *use* of the site, such as photographs embedded in a post by an editorial team, or an avatar uploaded by a user. While the context may differ, the end goals are the same: automated encoding and compression based on settings defined by the development team.

Fortunately, the image processing libraries you've come to understand from your local development workflows can be used in any number of contexts. And while there can never be a one-size-fits-all approach to your responsive image markup, these systems provide sensible defaults, configuration options, and API hooks to ease their implementation.

Static Site Generators

Compared to task-runners, there's some similarity in the way static site generators such as Jekyll or Eleventy approach images. Using these tools to produce a deployment-ready website requires management of assets, including CSS minification or transpiling and bundling of JavaScript. As you might imagine, this means these tools enable you to process image assets the same way, using many of the libraries you've already learned about.

The official image plugin for Eleventy uses Sharp to provide resizing, generation of multiple source sizes, re-encoding, and compression, just like some of the tasks you've learned about here.

Unlike a task-runner, a static site generator has direct insight into both the configuration and usage of those libraries, and the markup being generated for the production site—meaning it can do a great deal more to automate our responsive image markup. For example, when invoked as part of a shortcode for displaying images, this plugin will output HTML according to the configuration options passed along to Sharp.

```
const Image = require("@11ty/eleventy-img");
module.exports = function(eleventyConfig) {

async function imageShortcode(src, alt, sizes="100vw") {
  let metadata = await Image(src, {
  formats: ["avif", "webp", "jpeg"],
  widths: [1000, 800, 400],
  outputDir: "_dist/img/",
  filenameFormat: function( id, src, width, format, options ) {
      const ext = path.extname( src ),
        name = path.basename( src, ext );

      return `${name}-${width}.${format}`
  }
  });

  let imageAttributes = {
  alt,
  sizes,
  loading: "lazy"
  };

  return Image.generateHTML(metadata, imageAttributes);
}

eleventyConfig.addAsyncShortcode("respimg", imageShortcode);
};
```

This shortcode could then be used in place of the default image syntax:

```
{% respimg "img/butterfly.jpg", "Alt attribute.", "(min-width: 30em) 800px, 80vw" %}
```

If configured to output multiple encodings, as above, the generated markup will be a `<picture>` element containing corresponding `<source>` elements, `type` attributes, and `srcset` attributes already fully populated with a list of generated candidate sizes.

```
<picture><source type="image/avif" srcset="/img/butterfly-400.avif 400w, /img/butterfly-800.avif 800w, /img/butterfly-1000.avif 1000w" sizes="(min-width: 30em) 800px, 80vw"><source type="image/webp" srcset="/img/butterfly-400.webp 400w, /img/butterfly-800.webp 800w, /img/butterfly-1000.webp 1000w" sizes="(min-width: 30em) 800px, 80vw"><source type="image/jpeg" srcset="/img/butterfly-400.jpeg 400w, /img/butterfly-800.jpeg 800w, /img/butterfly-1000.jpeg 1000w" sizes="(min-width: 30em) 800px, 80vw"><img alt="Alt attribute." loading="lazy" src="/img/butterfly-400.jpeg" width="1000" height="846"></picture>
```

Of course, this plugin won't be able to *generate* a viable `sizes` attribute, as it can't know the ultimate size and position of the image in the rendered layout, but it does accept one as input when generating your markup—another job for RespImageLint.

Frameworks

Client-side rendering frameworks will require a task-runner or bundler like Webpack to edit, encode, and compress image assets themselves. Responsive-loader, for example, also uses the Sharp library to re-save image assets. It then allows you to then `import` your images as objects:

```
  import imageAVIF from 'img/butterfly.jpg?sizes[]=400,sizes[]=800,sizes[]=1000&format=avif';
  import imageWebP from 'img/butterfly.jpg?sizes[]=400,sizes[]=800,sizes[]=1000&format=webp';
  import imageDefault from 'img/butterfly.jpg?sizes[]=400,sizes[]=800,sizes[]=1000';
```

These imported images can then be used through abstractions like React's Image component, or to populate your responsive image markup directly:

```
<picture>
  <source type='image/avif' srcSet={imageAVIF.srcSet} sizes='…' />
  <source type='image/webp' srcSet={imageWebp.srcSet} sizes='…' />
  <img
    src={imageDefault.src}
    srcSet={imageDefault.srcSet}
    width={imageDefault.width}
    height={imageDefault.height}
```

```
    sizes='…'
    loading="lazy"
  />
```

A framework that does client side rendering is a strong candidate for Lazysizes and `sizes="auto"` —giving you almost fully automated responsive images.

Content Management Systems

WordPress was one of the earliest adopters of native responsive images markup, and the API has been gradually improved since being introduced in WordPress 4.4 with support for WebP and control over the output mime type. WordPress core is designed to make use of the ImageMagick PHP extension (or, absent that, the GD library).

When an image is uploaded through the WordPress admin interface, that source image is used to generate user-facing files on the server, in much the same way as you would on your local machine. By default, any image output by WordPress will come with a generated `srcset` attribute based on the image sizes configured in your theme.

Two key settings that can be configured for generated images are the compression quality and the output mime type.

For example, to set the default compression quality to `70` for all generated images, use the following:

```
add_filter( 'wp_editor_set_quality', function() { return 70; } );
```

For even better compression, switch the output format for uploaded JPEG images to WebP with the following:

```
add_filter( 'image_editor_output_format', function( $mappings ) {
  $mappings[ 'image/jpeg' ] = 'image/webp';
return $mappings;
} );
```

Given that WordPress has full understanding of all alternate cuts and encodings it generates from an uploaded image, it can provide helper functions like `wp_get_attachment_image_srcset()` to retrieve the full, generated `srcset` value of an image attachment.

As you'll likely have guessed by this point, working with the `sizes` attribute is a little more fraught. Absent any information about how images will be used in a layout, WordPress currently defaults to a `sizes` value that effectively says "this image should occupy 100% of the available viewport, up to the largest source's intrinsic size"—a predictable default, but not a correct one for any real-world application. Be sure to make use of `wp_calculate_image_sizes()` to set contextually-appropriate `sizes` attributes in your templates.

Of course, there are countless WordPress plugins dedicated to making modern image workflows faster for development teams and users alike. Perhaps most excitingly, plugins like Jetpack's Site Accelerator (formerly "Photon") can provide *server-side* negotiation for encodings, ensuring that users will receive the smallest, most efficient encoding that their browser is able to support without the need for `<picture>` and `type` markup pattern. It does this through use of an image content delivery network—a technology you can make use of yourself, independent of your CMS.

All of this is also true of hosted CMS solutions like Shopify, though the mechanisms themselves will differ somewhat: offering similar hooks for generating alternate image sources and corresponding `srcset` attributes and art direction through the `<picture>` element.

□□□ #2
□ □□□ □□□ 2026-10-31 22:52:32 UTC
□ □□□ □□□ 2026-10-31 22:52:36 UTC