Image compression, resizing, and the details of optimising images for websites including art websites

Antsstyle (@antsstyle)

antsstyle3@gmail.com

May 22, 2023

Contents

1 Introduction

2 Image formats, encoders, and compression

For the purposes of this article, we will be talking about non-animated 2D images. These can be stored in several different formats, using different encoders and/or different compression types, as explained below.

2.1 Lossy compression

Lossy compression means that to achieve a better compression result (i.e. a smaller filesize), some of the image data is permanently lost. This is a deliberate tradeoff; lossless compression, while discarding no image data, generally cannot achieve the same level of file size reduction.

In most cases, a lossily compressed image will have a quality loss so small that the viewer will never notice it. That being said, overly compressed images have an infamous look to them...

slitherpunk Follow

exports you as a jpeg *and turns the quality waaaaay down*

devipotato Follow

thanks. i really needed that

dead-god-dreaming Follow

Not a legitimate basis for a system of image optimisation.

2.1.1 Chroma subsampling

One prominent aspect of image compression involves the technique called chroma subsampling. This means reducing the range of possible colours in an image, effectively changing the colour scheme slightly. This can be very beneficial; the human eye is bad at noticing certain colours, meaning that chroma subsampling can frequently improve compression performance without looking much different to a viewer. Old analog TVs using NTSC and PAL relied on this method for example.

Most people rejected His message.
this image only needs black and white
Shut up!
They hated Jesus because He told them the truth.
Gal. 4:16
15

He has a point there.

A black and white image can benefit greatly from chroma subsampling, since often the removed colours will have almost no impact whatsoever (e.g. random shades of gray not easily distinguishable from other shades of gray). Most image compression algorithms perform some level of chroma subsampling by default, although art websites will naturally want to disable this and preserve all colours.

2.2 Encoders

An image format, like JPG, is not one single format that is produced one way. An encoder is used to transform a file into a JPG; different encoders have different performance. For example, MozJPEG is a newer JPG encoder that achieves lower file size for the same quality as older encoders, but takes slightly longer to run.

Which encoder to use is important for a website. If an image is going to be resized a handful of times, but viewed by millions of users, then an encoder that runs more slowly but achieves lower filesize is going to be much better than one which is fast but doesn't have the same efficiency.

2.3 JPG

JPGs are the most common image format. JPGs are lossy; even a 100 "quality" JPG loses some of the original image data. JPGs have survived for so long because they are fantastic: they deliver solid image quality with small filesizes.

Note that the JPG "quality" parameter is not an actual percentage. It's an arbitrary number used to guide the compression algorithm: 90 quality does not mean 90% of the original image quality or anything else. The correlation is thus: a higher quality number equals better image quality, but also higher filesize. This correlation is **not** linear, and understanding it is crucial to understanding what quality settings to use.

Increasing the JPG quality parameter causes an exponential increase in filesize at higher values, and very little change in lower values. The quality increase is logarithmic: at the higher values, a change has less and less noticeable improvement in quality. Going from 60 to 70 quality results in a visible quality improvement with a fairly small filesize increase, whereas going from 85 to 95 quality results in a much smaller visual improvement with a large increase in filesize. This also means absurdly low qualities don't make much sense either; going from 30 to 20 image quality means a fairly low filesize decrease but a large decrease in quality.

As such, JPG quality suffers from heavily diminishing returns. Going above 90 quality is generally insane for website purposes: the user will never perceive a difference in image quality between 90 and 95 quality, but your bandwidth costs will absolutely notice the large increase in filesize. 80 quality is a common setting for small or thumbnail images that will not be displayed larger than 640x640, and 90 is a good setting for artworks at any size when being displayed on the web.

2.4 PNG

PNG is another old format that is still widely in use. PNG has one primary advantage over JPG: it can store transparency (alpha) values, making it useful for images that need to have a transparent background.

PNGs generally tend to be larger than JPGs, meaning that they are best when you actually need the transparency feature.

2.5 WebM

WebM is Google's media format for both images and videos (confusing, I know). It is specifically designed for use on websites; compared to old JPG encoders it has good filesize performance, but compared to MozJPEG the differences are insignificant (and worse for smaller images).

For art websites, this format should be avoided. Not due to specific weaknesses in the format itself, but because art software doesn't have a lot of compatibility with the WebM format (it is, after all, intended for web use). Makes it rather annoying for artists saving reference material only to find out WebM won't open in some programs.

3 Importance of image optimisation

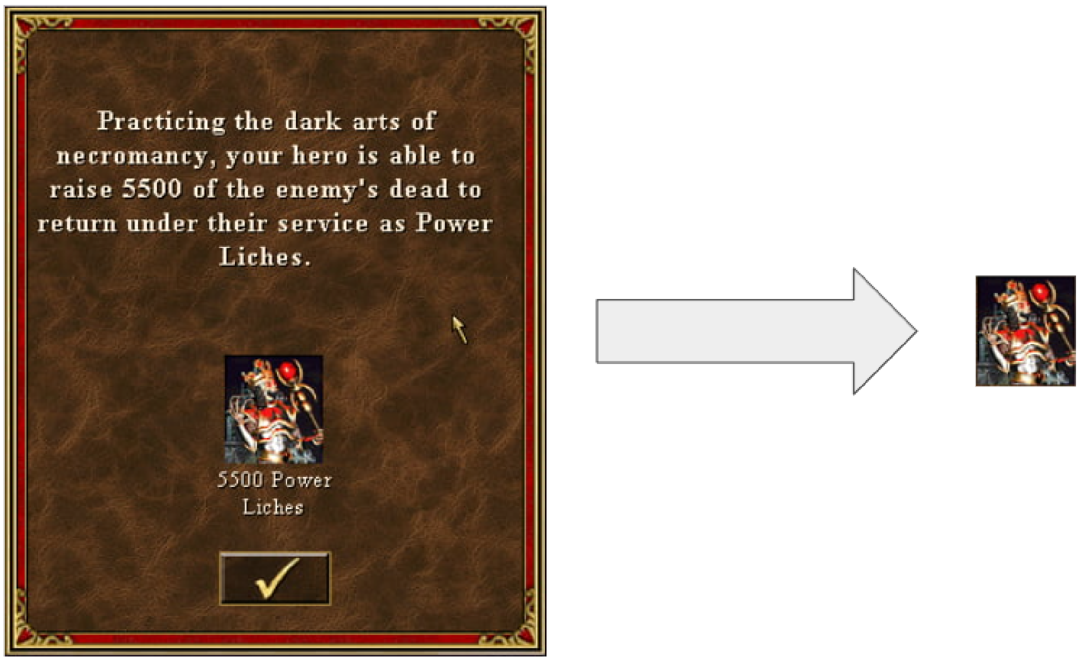
There are two primary reasons image optimisation is important for websites. Firstly, sending data to your websites' users costs money; the less data you have to send, the less money you will be paying for bandwidth. Secondly, if your website pages have less data in them, the user will be able to download and view them more quickly. For these reasons, most websites will resize images to several different sizes to be as efficient as possible when sending an image to a user.

3.1 Server-client relationship

A non-technical person, or one not too familiar with web development, might wonder: "why can't we just use one image size, and let the user's browser resize it for us?". The reason is fairly simple: no matter how the image is being displayed in the user's browser, it must still download the full image before it can display it. Let's look at two different examples.

3.1.1 Example 1: cropping a profile picture

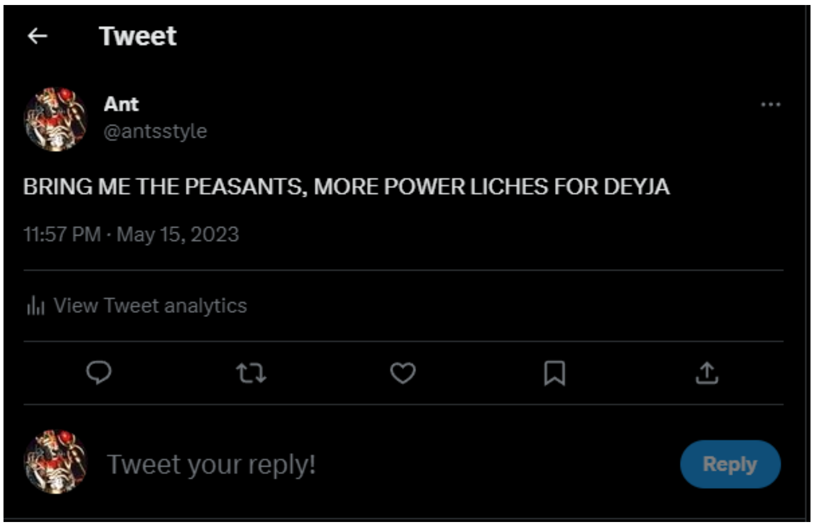
Suppose you want to change your profile picture on a website. You upload the image to the website and select the area of the image you want to be used as your profile picture. This means much of the image file isn't actually needed and can be discarded once the necessary area is cut out (cropped).


Practicing the dark arts of necromancy, your hero is able to raise 5500 of the enemy's dead to return under their service as Power Liches.

5500 Power Liches

The right image is approx. 10 times smaller in filesize than the left image.

If the server simply kept the full image, and cropped the image in some client-side code (such as in the user's browser) when the webpage was loaded, it would still have to send the full image to the user, even though only a tiny area is actually being displayed.


Tweet

Ant
@antsstyle

BRING ME THE PEASANTS, MORE POWER LICHES FOR DEYJA

11:57 PM · May 15, 2023

View Tweet analytics

Tweet your reply!

Reply

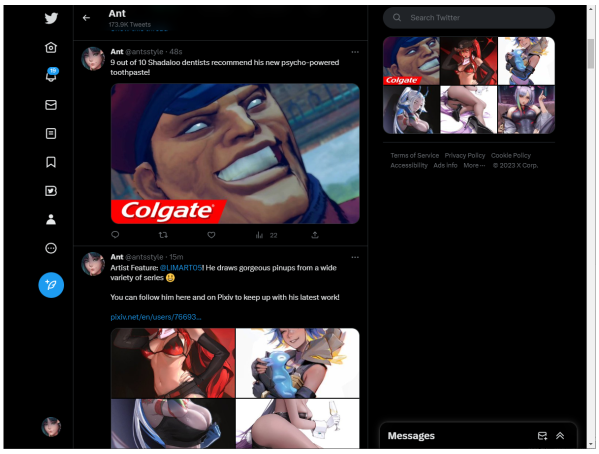
Sandro would be pleased.

Here is an example of the profile picture in action. If the full picture is being sent by the server, and subsequently cropped by the viewer's browser, that means the server is sending 10 times more image data than it needs to send to actually display this image. In functional terms, to the user, this will be an almost invisible inefficiency: unless they are still on a dial-up modem, they may not notice the large size of the image. It may not even be noticeable to the website admin, who may only notice that their bandwidth costs seem to be higher than usual.

For larger images, this problem can be hugely magnified, and result in both slow loading times for users and high bandwidth costs for websites.

3.1.2 Example 2: Displaying a large image in a small area

Suppose we want to upload an image to a website, such as a social media or art website.



Ant
173.9K Tweets

Ant @antsstyle · 48s
9 out of 10 Shadaloo dentists recommend his new psycho-powered toothpaste!

22

Ant @antsstyle · 15m
Artist Feature: @LIMART05! He draws gorgeous pinups from a wide variety of series 😃

You can follow him here and on Pixiv to keep up with his latest work!

pixiv.net/en/users/76693...

Search Twitter
Colgate
Terms of Service Privacy Policy Cookie Policy
Accessibility Ads info More ··· © 2023 X Corp.
Messages

Resized image on the left is displayed on the webpage at a resolution of 450x253 pixels; the full image displayed on the right is 1920x1080 pixels at its full size.

In this situation, the full artwork is displayed (i.e it has not been cropped), but it has been resized to smaller dimensions and a smaller file size.

4 A basic infrastructure overview

Uploading an image to an art site, or social media, involves more than just your computer and a web server. There are several components at work:

- **Your device**: the device or computer you upload the image from.
- **Web server**: a web server for the website you're using, that your computer interacts with. Your image will first be uploaded here, then sent for processing.
- **Cloud storage**: a storage area for files, not unlike Dropbox or a similar service, but designed for use in code rather than a user interface. The web server will upload your image here so that serverless code can process it.
- **Serverless code**: this is used to get the image file from cloud storage, resize it, and put the new resized versions in cloud storage as well.
- **Content Delivery Network (CDN)**: once the images are ready to be displayed, they are loaded from a CDN, which gets the image file from cloud storage and then caches it for fast retrieval later.

The subsections below give a basic idea of the execution flow for uploading an image to a web server, the infrastructure involved, and why it is necessary to have all this extra stuff instead of a more simple system.

4.1 Webserver load

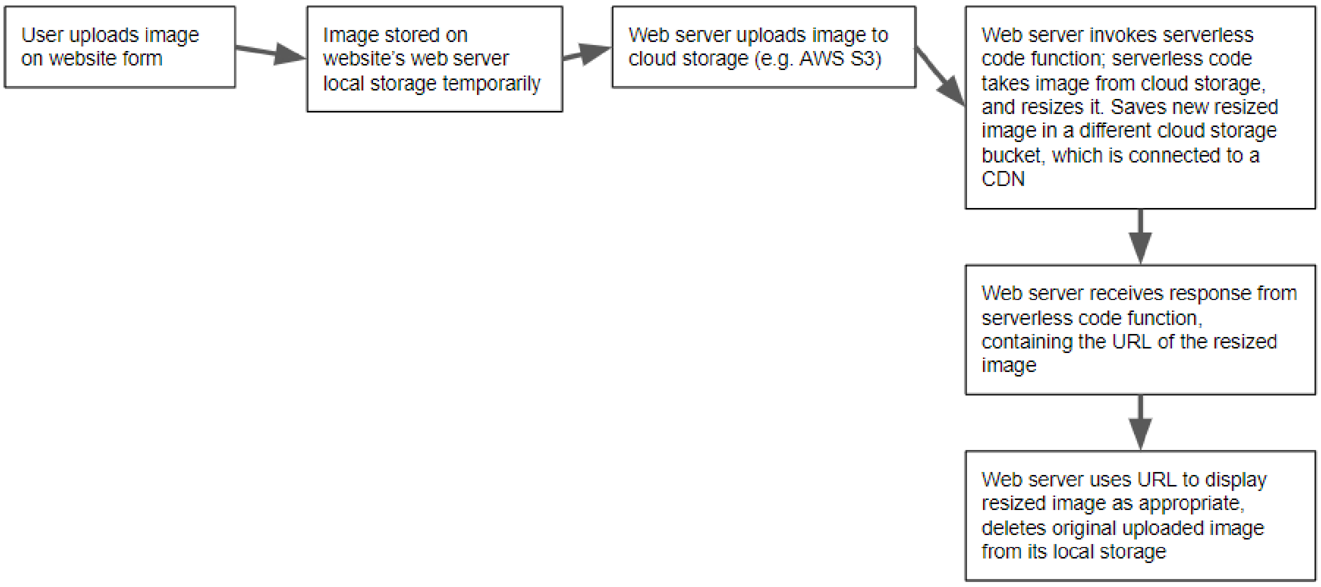
Resizing images on webservers themselves, instead of via a cloud service, is generally a bad idea. Firstly, webservers aren't optimised or suited for heavy processing tasks: their job is to serve web requests, and if they have to resize images, it will slow them down in that task considerably. Secondly, it will take them a while to resize the images, causing the user to have to wait either at the submission screen or when the page loads. This is where serverless code comes in very handy.

4.2 Serverless code

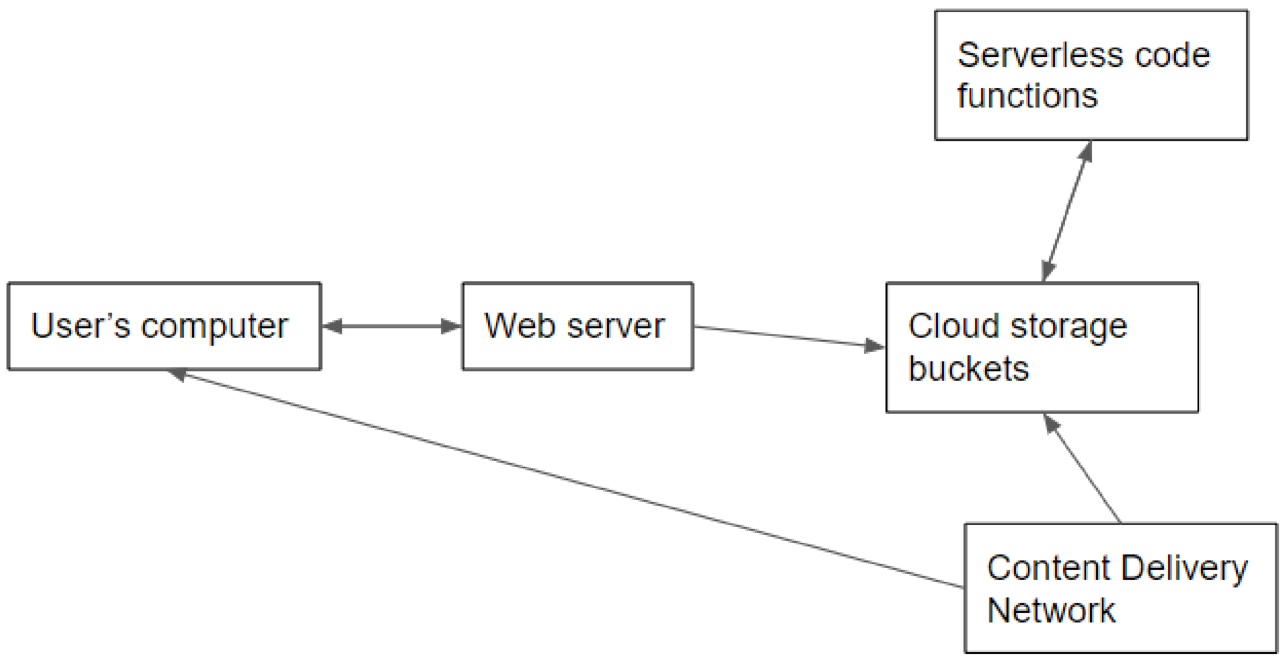
Despite the name, “serverless” code isn’t actually serverless; after all, it must run on *something*. It means that you haven’t provisioned your own server to run the code; the web service provider, e.g. AWS or Microsoft Azure, will have a pool of servers that are ready to run serverless tasks for customers. When you invoke your code or function, the web service provider assigns one of its servers to execute it.

This is very useful for image resizing: maintaining a dedicated server would be wasteful as it wouldn’t always be needed, and because web service providers have a huge capacity for serverless tasks and allow users to run many concurrent tasks, it scales easily. We can also utilise this high capacity to perform several resizes concurrently for the same image, further reducing the time needed for all image resizes to complete.

4.3 Example flowcharts


User uploads image on website form
Image stored on website’s web server local storage temporarily
Web server uploads image to cloud storage (e.g. AWS S3)
Web server invokes serverless code function; serverless code takes image from cloud storage, and resizes it. Saves new resized image in a different cloud storage bucket, which is connected to a CDN
Web server receives response from serverless code function, containing the URL of the resized image
Web server uses URL to display resized image as appropriate, deletes original uploaded image from its local storage

A hastily-made flowchart of sorts, showing the basic flow for resizing an image when the user uploads it.



A basic representation of the infrastructure that is in action when resizing a user's image.

5 Infrastructure considerations

5.1 Serverless code performance

Serverless code, because it's not running on a specific server, has to be configured so the web service provider knows how much power you want your code to use. This affects the cost of running the code, and also the amount of time it will take. Generally, serverless code is charged in GB-seconds or similar measures (basically, you are charged for how much processing power you asked to have assigned to your code, and how long it had to run for).

This means using more power can be worthwhile in many cases, where the runtime decreases enough that the cost barely increases (since you are using more power but for less time). In addition, serverless code is fairly inexpensive to use, and so for cases like image resizing where the user has to wait for it to complete, using maximum power settings is often a good idea to minimise waiting times.

In AWS Lambda, you can use the Configuration – General Configuration tab to set the amount of memory your function has allocated to it, which also changes the processing power allocated to it (AWS allocates processing power proportional to how much memory you assign, if I remember correctly, 1769MB of memory equals 1 vCPU). Setting the maximum of 10240MB for image resizing is a good idea, as the cost will be minimal and it will minimise the time needed to resize very large images. You may also need to increase the timeout period to be 10-15 seconds, just in case of a very large image; the default is sometimes too short.

5.2 CDN: Content-Disposition headers

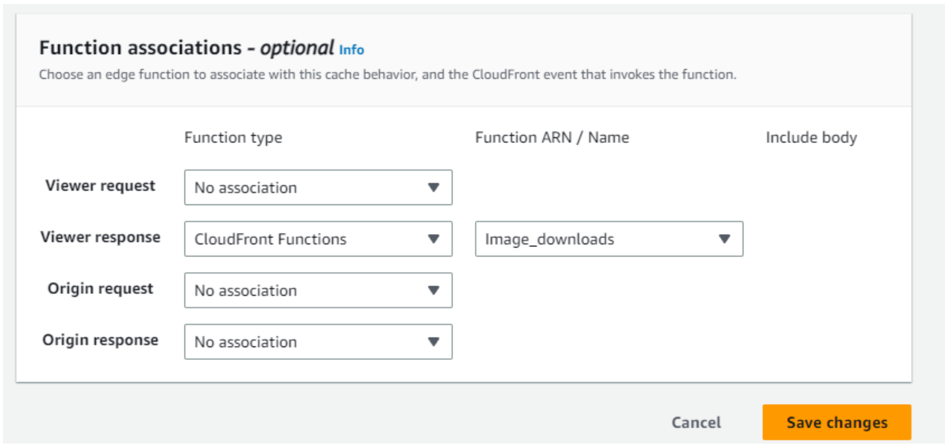
Many websites that allow images to be viewed also have a convenient "download" button, allowing a user to download a larger version of the currently displayed image when appropriate. Enabling a download button to bring up a browser prompt, and not simply display the larger image directly in the browser, requires a little adjustment to the CDN you are using.

Bringing up a download prompt in the browser (e.g. a 'save file here' window) requires setting the HTTP

Content-Disposition header in the response to the user, so that the browser will interpret the response as a download. An example of how to do this in Amazon CloudFront is below, but the same method and principles should apply to most web services providers.

Firstly, you need to make a CloudFront function, or equivalent for your provider. The point of this function is to examine incoming requests, and add a content disposition header if two conditions are met: the request is for an image file, and it contains a query string signifying the user wishes to download the file, such as "dl=1". Sample code for this function can be found in the sample code repository for this document here.

In your CDN distribution settings, you need to tell CloudFront to use this function for your CDN, as shown below.


Function associations - *optional* Info

Choose an edge function to associate with this cache behavior, and the CloudFront event that invokes the function.

	Function type	Function ARN / Name	Include body
Viewer request	No association		
Viewer response	CloudFront Functions	Image_downloads	
Origin request	No association		
Origin response	No association		

Setting our function to be used for 'Viewer Response', which means responses sent back to your users from the CDN. 'Origin' in this context means your cloud storage that the CDN retrieves files from.

With these two steps completed, you can now cause urls from your CDN that point to image files to bring up a download prompt if they have the appropriate query string at the end of the URL, like so:

```
https://cdn.some-image-website.com/images/640x480/animagefile.jpg?dl=1
```

5.3 Lazy and eager evaluation

In computer science, "lazy" evaluation means something that is only evaluated when it has to be. For example, suppose that today is a Wednesday, and that it is the most important day in your life. If we then evaluate this statement:

```
if (today is a Tuesday AND it is the most important day in your life) {
   achieve world domination
}
```

The condition "today is a Tuesday", in our hypothetical scenario, evaluates to **false**. Because this is an AND statement, whatever condition comes after makes no difference: it doesn't need to be evaluated because the whole statement will be false no matter what. This is an example of lazy evaluation; eager evaluation is when everything is evaluated, even if it doesn't have to be.

A similar concept exists for how to implement image resizing for a website. We can either be 'lazy' - only resizing an image when someone tries to view it - or 'eager', performing all necessary resizes as soon as it is uploaded. Each is detailed below, but for art sites where the image is expected to be viewed at least once, **eager** evaluation is usually preferable.

5.3.1 Lazy evaluation

For the purpose of image resizing, lazy evaluation means that the image is only resized when first requested. The URL for that image might look like this:

`https://cdn.some-image-website.com/images/640x480/animagefile.jpg`

A serverless code function can then resize the image to the dimensions specified in the URL as parameters. There is an important pitfall to avoid here: the resizing code should have a specific list of resizing values that are allowed, or else a malicious user or a carelessly executed use case could cause your image to have loads of resized versions in sizes that nobody needs (e.g. 640x479, 640x478...), eating up storage costs or causing your web pages to display images poorly.

Lazy evaluation has one advantage: it can result in slightly lower storage costs (since an image that is never viewed will not be resized), and for smaller images being shown at low resolutions, the time needed to resize the image for the first time when a user tries to view it will be almost unnoticeable. It's also a little easier to do.

I wouldn't recommend it in most cases, as for any image that you expect to be viewed (including, for instance, by the user after uploading to confirm it was uploaded successfully), the resizing will be necessary no matter which method you choose; lazy evaluation in this case just leads to potential problems.

5.3.2 Eager evaluation

Image resizing using eager evaluation means that, at the time of the user uploading the image, we perform all necessary resizes immediately. This avoids the first view website visitors who try to view the image having to wait for a resized image to be processed before it can be displayed, which for larger images like big artworks, can take a few seconds. Another advantage is that if there is an error resizing the image, we can immediately tell the uploading user about that error, instead of a website visitor getting a broken image and the user only finding out later when someone complains.

Finally, there is a slightly more complicated technical advantage. When a user is uploading an artwork on an artwork submission form, showing the full image file they uploaded in the submission form is a minor security issue; the only way they can be shown the full image is if it is publicly available on the internet. For an art website, the user doesn't often want the full, unmodified image file they uploaded being publicly available; if someone were to guess the URL of the image before the web server deletes it, they would have access to a file the user didn't want them to have access to. In practice this vulnerability is extremely hard to exploit in any useful way, but it's good to remove it nonetheless.

The main downside of eager evaluation in this context is that because we must perform all resizes immediately, the uploading user may have to wait a few seconds at the submission form before we can confirm the resizing was successful. For small images this length of time will be almost unnoticable, but resizing very large images can take a few seconds.

6 Client-side considerations

6.1 User screen size and orientation

For normal viewing purposes, an image never needs to be bigger than a user's potential screen resolution. For example, if you have a 3000x5000 pixel image, users on widescreen 1080p (1920x1080) monitors will be seeing it as 648x1080 resolution filling the height of the screen, and users on portrait orientation devices like smartphones will be seeing it at 1080x1920 or so.

Since a user might view a website on either device, an image intended for full screen viewing would therefore want to be resized to a maximum of 1920x1920, depending on its aspect ratio.

6.2 Image quality requirements

Not all images need to be in super high quality. For example, images that will only displayed at small sizes will be indistinguishable at lower quality due to their smaller size, and images that function as previews or don't need to look amazing will work fine at lower qualities as well. Deciding what quality settings to use depends on a few factors, which stack with each other somewhat.

Images that will generally be skipped over or scrolled past such as images on tweet previews, do not need to be high quality; since the user is not going to be looking carefully at them, they will not notice a lower quality on such an image. Images that will only be displayed in fairly small resolutions (e.g. thumbnails at 640x480) in a user's client will look mostly fine at lower quality settings. Finally, the image use case is also relevant: artworks that don't need high quality settings, for instance stock photos on a business website, will usually be fine with lower quality settings.

7 Sample code

7.1 Libraries

Not all software libraries for manipulating images are written with performance as a top priority; some prioritise ease of use, graphical user interfaces, or other considerations. For a website hosting a lot of images, performance - i.e. speed for a given operation, compression efficiency, resulting image quality - is by far the most important consideration.

Libraries that often come with web servers, like Imagick, are ill-suited for this kind of task. I would recommend using anything that is powered by the open-source library libvips, such as Sharp on the Node.js platform. For the example code below, we will use Sharp.

7.2 Compiling Sharp.js for use

Sharp uses libvips to work, and because of this, it is not platform-independent like some other JavaScript packages. It must be compiled on the platform you want to use it on; for example, running it on AWS requires compiling it on an x86_64 platform, such as a Node.js Linux server.

For convenience, I have pre-compiled the Sharp library on x86_64, version 0.30.7, which is available in the GitHub repository. To compile other versions or on different architectures, you will need to set up a machine on that architecture, install the Node Package Manager (npm), then run **npm install sharp**.

7.3 Sample code for resizing an image using Sharp.js

Sample code for resizing an image via Sharp on Amazon Web Services can be found in the GitHub repository for this document here: https://github.com/antsstyle/website-img-compression-explanation. There are some additional explanations in the code file to illustrate what it does.