# How we optimized the performance of our website

Web caching rules used at InstantLuxe.com


**Introduction** **3**

**Guidelines** **4**

Rationale 4

Examples 4

Images 6

Other static assets 7

**Front-end** **8**

Protocol 8

Media 9

Lazy-loading 9

Use the dominant color as the background 10

Use a low-res image as a placeholder (blur-up) 10

Web server config 11

httpd 11

nginx 13

Serving text files 14

Serving media files 14

Benchmarking the homepage 17

Network latency 18

**Scripts** **20**

Setting up 20

Using a temporary virtual machine for the tests 20

Server performance tips 21

Use the RAM to store temporary files 21

Use a functional approach for scripts 22

A word on “find” 22

Installing the binaries 23

About the perceptual difference with image encoding 24

Composing the toolchain 25

About temporary files 25

Wrappers 26

The launcher script 28

Committing the new file 29

Putting it all together 30
Improving even further 30

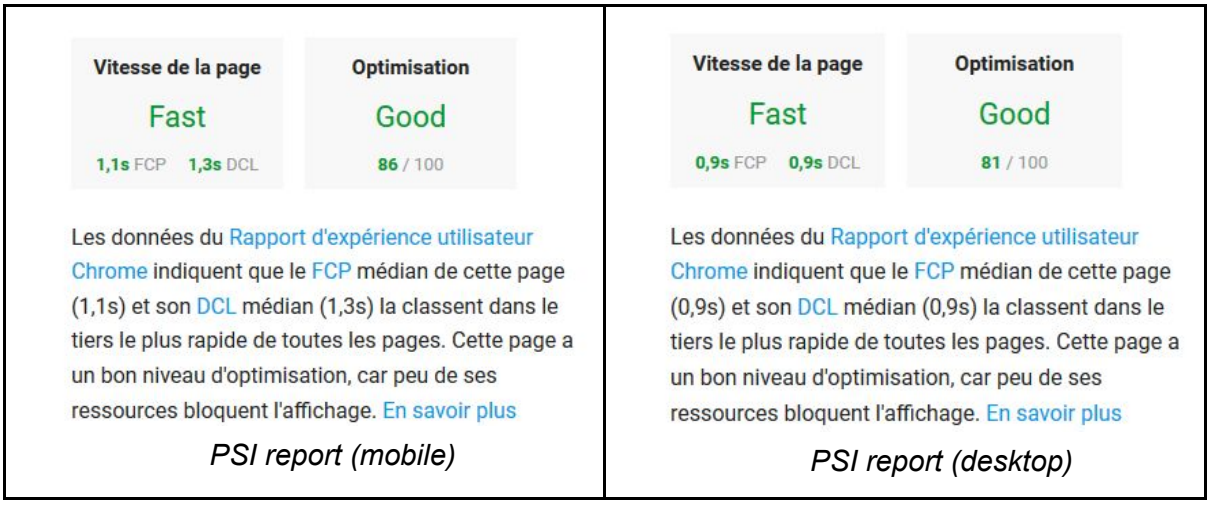
# Introduction

By simply following best practices and using good open source tools, not necessarily costly services, here is what you could achieve:


*Performance summary with WebPageTest[1] (May 31st, 2018)*

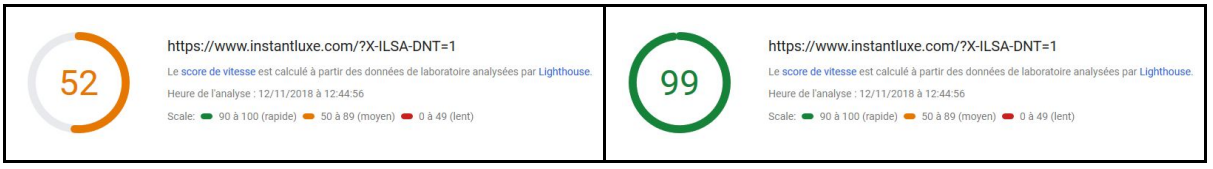
We can reach several thousand users at the same time without a notable slowdown, which is well beyond our usual usage. Since our users are mostly located in France, our servers are there as well and we don't have much use for a CDN (yet). Likewise, we don't use serverless or a load balancing system, this is served by a only few dedicated servers with simple roles.

Another view is reported by PageSpeed Insights[2] with Chrome UX Report[3] (May 31st, 2018):

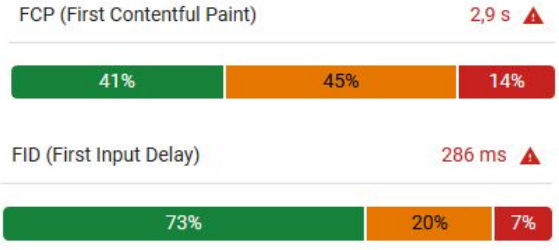



*PSI report (mobile)* | *PSI report (desktop)*

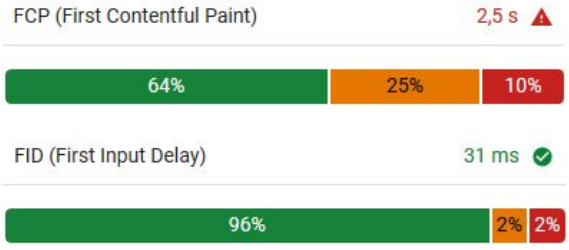
Or the more recent[4] Chrome UX Report (November 12th, 2018):




[1] https://www.webpagetest.org/
[2] https://developers.google.com/speed/pagespeed/insights/
[3] https://developers.google.com/web/tools/chrome-user-experience-report/
[4] https://webmasters.googleblog.com/2018/11/pagespeed-insights-now-powered-by.html




*PSI report (mobile)*




*PSI report (desktop)*

What follows can be seen a set of guidelines, tips and tricks that we have used to improve the technical performance of our website. It may or may not apply to anyone else, or have the same results if you do.

Modern image optimisation tutorials on the web tend to use JavaScript as a base tech in their tooling. We have been to use Bash for the most part. To follow along, you will need to be able to install new software on a machine. Since we are talking about web performance, a web server is required as well (examples are provided for both **httpd** and **nginx**).

# Guidelines

## Rationale

There are many ways to optimise performance. You need a culture of performance, you can't just apply a few tricks and be done, otherwise you may find someone uploading a video in the middle of a blog piece. You need to keep checking that what you have set up stays in place, and you have to advocate for performance with your team. There are ways to improve performance that only technical people will be able to solve (new compression formats), and there are others that have little to do with technical skills (clean up old campaigns in your Tag Manager of choice).

The one big win is "efficient caching rules". We are talking about web requests and responses here, and browsers have had caching strategies for years if not decades. We may just as well leverage them. Once you have improved that caching, you have improved navigation for all requests after the first load. Ideally, all your users would load more than one page so this is the biggest low hanging fruit.

Simply put, every asset should be made "immutable" by its unique URL, and new versions of the asset should be reflected by a change in its public URL.

## Examples

The idea is to give every asset its own unchanging URL and to tell the browser to please cache the asset for as long as it can. Asking for a different compression format or image

codec won't affect the URL, because essentially you are still asking for the same content and what you already downloaded is good enough. What changes between these requests is the context, usually in the form of HTTP headers like “Accept”.

Request InstantLuxe’s logo without context (the most widely supported format):

```
$ curl -I "https://assets.instantluxe.com/images/logo/instantluxe.png"
HTTP/2 200
server: nginx
date: Tue, 12 Jun 2018 08:32:04 GMT
content-type: image/png
content-length: 5043
last-modified: Thu, 04 May 2017 14:59:45 GMT
etag: "590b41e1-13b3"
expires: Thu, 31 Dec 2037 23:55:55 GMT
cache-control: max-age=315360000
link: <https://assets.instantluxe.com/images/logo/instantluxe.png>; rel="canonical"
vary: Accept, CH-DPR
cache-control: no-transform, public, immutable
accept-ranges: bytes
```

Request the logo with a browser that supports the WebP image format:

```
$ curl -H "Accept: image/webp" -I
"https://assets.instantluxe.com/images/logo/instantluxe.png"
HTTP/2 200
server: nginx
date: Tue, 12 Jun 2018 08:32:21 GMT
content-type: image/webp
content-length: 4222
last-modified: Thu, 04 May 2017 14:59:45 GMT
etag: "590b41e1-107e"
expires: Thu, 31 Dec 2037 23:55:55 GMT
cache-control: max-age=315360000
link: <https://assets.instantluxe.com/images/logo/instantluxe.png>; rel="canonical"
vary: Accept, CH-DPR
cache-control: no-transform, public, immutable
accept-ranges: bytes
```

Another structuring guideline with assets (not necessarily with dynamic documents) is to try our best to serve the content, even if with different parameters, so long as it is essentially the same. For example if you request the same logo with a format that does not exist, say “`Accept: image/xyz`”, you will get the default format instead (not an error, because those tend to be HTML documents). The idea is that our images are intended for use in an HTML `<img>` tag, where browsers don’t much care that the HTTP request header matches the URL or the actual codec used for the image. So long as the resource can be presented as an image, it will work. Later in this document, we will get to the server config to achieve this easily. It works with stricter APIs too, because the HTTP response includes the content type of the file that the server shipped.

Now let’s say you browse the website and the browser caches a bunch of assets. Say one of these happens to be a non-optimised version, because you were browsing while the optimisations were taking place. Your browser may spend a little bit more time downloading the full size version of this asset, but now it has cached it with a long lifetime like any other asset. Later, when the optimised version of that asset is available to the rest of the world and

you continue your browsing, your browser won't even ask for the download because that would waste even more bandwidth. Your locally cached version is already good enough, but if your browser did make the request, it would get the optimised asset.

However, whenever we update an asset on disk, we change its version number in our system so the HTML references a new URL (*ie*. instantluxe.1.png, instantluxe.2.png, instantluxe.3.png *etc*.). This way, browsers' caching strategies discover a new URL, are unable to find a local match and they need to fetch the asset again. The old version of the asset will never be used again, so eventually its "last used" timestamp in the browser's caching system will become stale and the browser will prune old entries.

## Images

With images making up the most of the content downloaded from any modern web page (except audio and video, but let's not go there), they are going to be our primary focus in this writeup.

The best way we have found to optimise images is to re-encode them in several alternate formats (WebP, JP2, JXR…), and also keep a re-encoded copy in their own format (metadata strip, better compression…). Each image file is made available in several formats so that when the web browser requests its preferred format, the server is able to match against simple content negotiation rules to ship the best file.

The alternate formats have a narrower browser support than the standard JPEG/PNG/GIF: they are WebP (Chromium-based), JPEG-2000 (Safari) and JPEG-XR (IE, Edge). Each of our images is passed through scripts that try to re-encode their thumbnails to each of these formats, and keep only versions when the file is smaller than for the standard format.

For instance, the logo from our parent company is a 8.6 Kio PNG file with a 6.4 Kio JP2 and a 3.1 Kio WebP variants. However, our own logo is a 8.7 Kio PNG file with a 7.3 Kio WebP variant and no JP2 variant.

Whenever that makes sense, we use SVG files instead of actual image files. For instance, the logos for our rental classification (black, gold, silver and ultra) weigh around 1 Kio for the source SVG, but around 470 Kio when Gzipped and usually less than 400 Kio with Brotli.

For more details on image optimisation techniques, you can look up Addy Osmani's eBook[5] and Ilya Grigorik's guide[6].

Images are served depending on browser support and expected gains. In order, we will prioritize WebP first, then JP2/JXR and finally JPEG/PNG, with fallbacks to the non-optimised thumbnail and finally the full-size image.

---

[5] https://images.guide/

[6] https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization

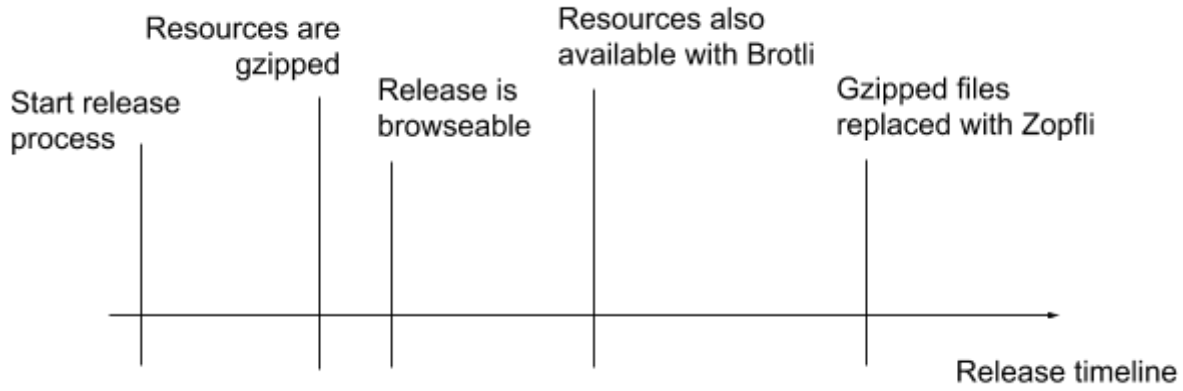
## Other static assets

After images, the next biggest chunk of data sent down the wire are static assets (mostly CSS and JavaScript files but also fonts).

Any file can be compressed using various technologies with variable file size gains and browser support. There are even people who advocate for gzipped images on top of better compression, but it is most effective with text-based files and with fonts.

In the case of CSS and JS files, they are also minified and post-processed before they are compressed. This gives us an opportunity to optimise their rules, maybe fix some compatibility issues or rewrite some parts in a way best suited for computers (as opposed to the humans who wrote them).

When we ship a new release, we compress each asset with simple GZIP at first, and then we replace this file with a better Zopfli encoding. We also compress with LZMA and Brotli.




*Availability of asset compression formats with regards to the release process*

For example, our own JS file is 223 Kio; minified, it becomes 139 Kio or 57 Kio once it is Gzipped; the final Zopfli file is 38 Kio while the LZMA is 33 Kio and the Brotli version is 31 Kio.

As we will see later, even some of our pages are “static assets” as far as the web server is concerned.

Static assets are served depending on browser support and expected gains. In order, we will prioritize Brotli first, then LZMA (although it has since been phased out since Safari started supporting Brotli) and finally Zopfli/GZIP, with fallbacks to the uncompressed minified file and finally the source file.

# Front-end

There is much to discuss here, but the main idea is to adhere to progressive enhancement in all things. Start small and iterate, then add to your build as time and browser support allow.

## Protocol

One big improvement was HTTP/2, in that we don't have to bundle our CSS and JS files to serve them with 1 request each.

Indeed, bundling files was an optimization of the network stack, when browsers needed to do one round-trip request and response call for every resource.

Before HTTP/2, the browser needed to request every asset over the network, and there was usually a limit of 6 concurrent resources per domain. However with the increase in complexity in recent years, websites have needed to include more and more CSS and JS files to be rendered properly. This meant that browser limits were reached (simultaneous downloads per domain), and the network stack began to slow down webpage rendering. To avoid this issue, developers started merging their assets into bigger CSS and JS files, bundles that were built for each page and served to the browser. But now the browser caching rules were inefficiently used, since the same libs were bundled under different URLs depending on the page that was served. There was no good solution.

Since HTTP/2, web pages only need reference their individual dependencies and let the software (browser + server) optimize the network calls. This makes it much simpler for everyone involved. HTTP/2 allows servers to send all the resources in one network stream, so the round-trip bottleneck is less of an issue. We can serve just the assets needed for the current page, and use efficient caching rules for each asset so the browser can decide whether each asset warrants a network call.

The one aspect of this protocol we have tried but that didn't work out like expected, was Server Push. Since browsers used to send uncredentiated requests, which means no "Accept" request headers, the browser often requested the same image with a WebP compatible header and also with a non WebP compatible header, both with the same URL. This means two things: the file is essentially downloaded twice, and the cache gets busted every time so there are even more downloads. Having all requests pass through Service Workers and add missing headers would likely have fixed this, but we didn't go that far.

We are aware that there is still much to do, especially splitting our own code into smaller files so that the browser doesn't have to download our whole custom library of features from the very first visit.

Take away: use & embrace HTTP/2 (mostly).

# Media

The biggest chunk of data you will likely serve to your users are media files. Audio files are not our specialty so we will not address them here. Video can be split into two categories, the ones that last over one minute (that we think are best served by a dedicated platform like Youtube), and the short animations that can be embedded with your content (like a loopy boomerang clip). Whether to autoplay and to include sound depends on your use case, and we won't address this concern here.

In our case, we only require static images, short soundless animations and the occasional embedded video. We allow autoplay for clips that are essentially a high quality GIF, and the rest is offloaded to another platform which usually waits for user interaction before doing anything.

The general layout of the page you want to optimise will dictate what kind of work that needs to be done. For instance if you have hand-crafted images to set an ambiance, you can spend some time to apply techniques you wouldn't probably have time for with user-submitted pictures. One example of this is to blur the background of the image, especially for the JPEG format, because the encoder will be more efficient.

On the other hand, if your page consists mostly of a big list of images, like a product listing, you probably won't be able to hand-optimize most images but you can apply other techniques. You need to focus on perceived performance, which is not necessarily the same as comparing raw end-to-end numbers. Experiments show that users prefer when pages load progressively, even if the overall time is longer, rather than being unable to display anything while the network is working and everything appears at once. This is especially true for the first visit, when the browser cache isn't primed yet and everything has to be downloaded over the network.

Users typically wait up to one second, sometimes three, until they lose patience and go away (there has been ample research on this by the likes of Google, Amazon). Think of it as a budget. If you are going to exceed this budget, and you should expect the first page load to do so because of the browser cache is not primed, then you will have to find ways to display things in chunks so the user stays focused.

Web browsers are keenly aware of this issue and they are spending a lot of engineering to improve both the software and the documentation for web developers. One of the advantages of the very first versions of Chrome was their multi-process architecture back in 2008, where all tabs were not using the same process and therefore one overzealous script could not halt the whole browser. Another big step was Firefox Quantum[7] in 2017: building on the multi-process idea, they now spread all tasks (download, parsing, layout *etc*.) over different threads so that these tasks can be processed in parallel without blocking the main

[7] https://www.youtube.com/watch?v=OwXLYoUj8J4 The future of the browser – Lin Clark – btconfBER2017

thread. On the developer relations side, Google keeps educating developers about yet other ways to split their code so that the browser can "ship a frame" more often[8].

Here are several ways to do this for a page with a long list of pictures. This list is not exhaustive.

## Lazy-loading

Basically, only load images that are on-screen or that are about to be displayed on-screen. Start with images displayed above-the-fold, then prioritize the others based on your knowledge of your pages. Maybe the images from the menu are next, or the images just below the fold, or wait for user action like scrolling, *etc*. Maybe you will want to load them all in a specific order, or trigger their loading with an event, or a combination of both (whichever comes first).

If your strategy is sound, this technique is usually without any downside. This is why Chrome is experimenting[9] with lazy-loading all images by default.

On the technical side, this is essentially what IntersectionObserver[10] was made for.

## Use the dominant color as the background

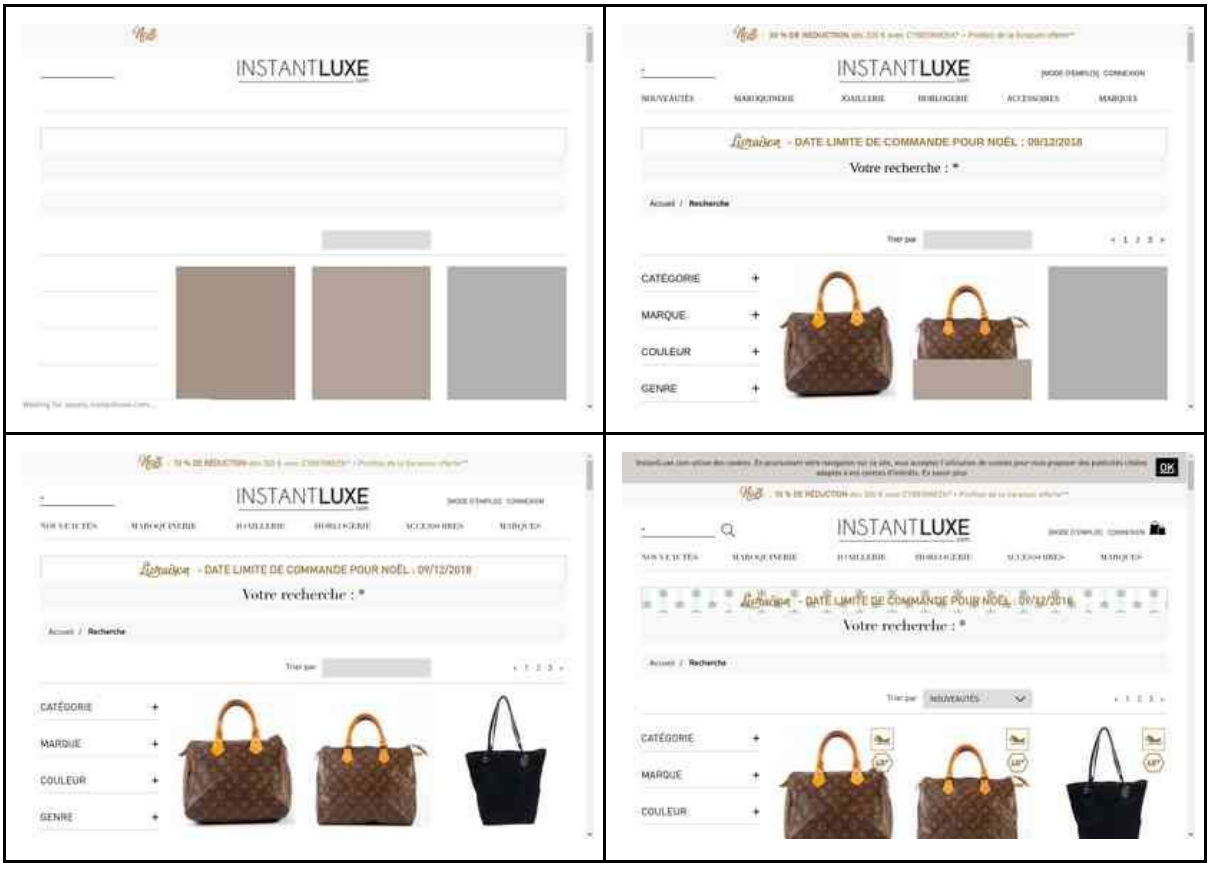
Identify the main color of each image on the back-end, then use CSS to include that as a background for this image (tip: an inexpensive way to obtain that color is to reduce the image to 1px*1px).

This will have a low impact on the overall weight of the page, but still include a variety of colors all over the page. It is especially useful for newsletters, where client software caching rules may be less effective than for web pages (if they even load your images), or mobile networks where we typically have to stare at blank pages for a long time and every bit of color helps.

The usual demo is Pinterest or a search with Google Image, or any of InstantLuxe.com's categories as well (shown below). To best observe this, use your browser's developer tools to throttle the network.

Here is an example filmstrip[11]:

[8] https://www.youtube.com/watch?v=Vg60lf92EkM Architecting Web Apps, Paul Lewis & Surma, Chrome Dev Summit 2018
[9] https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/czmmZUd4Vww/1-H6j-zdAwAJ
[10] https://developers.google.com/web/updates/2016/04/intersectionobserver
[11] https://www.webpagetest.org/result/181126_0W_d6654dbca936adc312deba74e4b4e62e/




Using this technique has a low impact on page size but the necessary indexing work may seem daunting. Another downside is when you also need strict Content-Security-Policy[12] or Subresource Integrity[13] rules preventing inline styles, in which case it becomes a bit tricky. An upside is that there is no JavaScript involved: this is the kind of optimization that wouldn't break other things even if it stopped working.

*NB: At the moment, this doesn't work well with Chrome (Canary), likely because of its better handling of HTTP/2 priorities[14] where above-the-fold images are loaded fast, so the background color is eclipsed early.*

## Use a low-res image as a placeholder (blur-up)

You can understand this either as a better way for the previous tip, or as basically another way to do progressive JPEGs: awareness of the network conditions, the device they are operating on and user interaction all become factors to determine the image quality best suited for each initial situation and at which point to move on to a better resolution.

The idea is to generate very low resolution of your images and place them as the images' URL for the initial page load, then dynamically fetch and swap the bigger resolution images.

---

[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/style-src
[13] https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
[14] https://blog.cloudflare.com/http-2-prioritization-with-nginx/ Optimizing HTTP/2 prioritization with BBR and tcp_notsent_lowat, Patrick Meenan, 2018

This way, the initial page load may have nothing (or just the background color) and then the smallest assets start arriving from the network. These low-res images are so small that they should arrive fast, especially with the help of HTTP/2 prioritization. At that point, users will see the outline of the image and they will have a general idea what to expect when the full image arrives. When that happens, you may want to apply a transition, perhaps morphing[15], depending on your need to attract the eye to the new image or not.

The usual demo is the first image of every Medium post.

There are several ways to go about building this feature, and various technologies with various degrees of browser support, so don't be afraid to experiment to find what works for your use cases.

Here are some of the technologies you could use here: Service Worker, IntersectionObserver, Custom Elements, image morphing.

Apart from the extra work needed to generate additional low quality thumbnails and their variants, there is also a considerable amount of work to build the logic to fetch and swap the images, especially considering efficient caching for the next page load. Because the goal is to reduce the total loading time of the page, fetching additional media from the network may be useful. In our case, the experiments didn't measure up to what we expected and more work would have been required.

[15] https://www.youtube.com/watch?v=tHJwRWrexqg "Supercharged Live! (Polymer Summit 2017)"

# Web server config

With the following config, we have been able to serve hundreds of thousands of requests per minute on the homepage, not counting the assets (the bottleneck seems to happen around half a million per minute, see below for the benchmarks). The network throughput was more of a bottleneck than any dynamic part of the system like (usually) the database, or even the caches and search index.

Requests that match predefined patterns don't go through any layers of the app; instead, the server streams a file directly from disk to the network (in our case, the PHP scripting language), thus avoiding the database entirely. Even that can be improved by storing these files in RAM.

## httpd

We have been using Apache 2.4 as a web server for the dynamic content but not for the assets. However, we have also kept pre-generated static copies of the most used pages and they were served by httpd as well.

For example, here are the cached variants for the homepage:

```
-rw-r--r-- 1 instantluxe www-data 69295 2018-11-19 16:00 index-fr.http2.html
-rw-r--r-- 1 instantluxe www-data  9228 2018-11-19 16:00 index-fr.http2.html.br
-rw-r--r-- 1 instantluxe www-data 11257 2018-11-19 16:00 index-fr.http2.html.gz (gzip)
-rw-r--r-- 1 instantluxe www-data 10784 2018-11-19 16:00 index-fr.http2.html.gz (zopfli)
-rw-r--r-- 1 instantluxe www-data 10186 2018-11-19 16:00 index-fr.http2.html.lzma
```

The static copies we keep of the homepage and a few other landing pages vary on:

- Language (fr, en, it, zh);
- ~~Protocol (http1, http2);~~
- With or without external assets, scripts and tracking (Do Not Track);
- Compression (none, gz, br, lzma).

The files can be served like follows:

```
# HTTP/1.x

## homepage w/ DNT

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} br
RewriteRule ^$ /.cache/index-fr.http1.dnt.html.br [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
```

```
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} gzip
RewriteRule ^$ /.cache/index-fr.http1.dnt.html.gz [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteRule ^$ /.cache/index-fr.http1.dnt.html [L]


## homepage w/o DNT

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} br
RewriteRule ^$ /.cache/index-fr.http1.html.br [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} gzip
RewriteRule ^$ /.cache/index-fr.http1.html.gz [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteRule ^$ /.cache/index-fr.http1.html [L]


# HTTP/2

## homepage w/ DNT

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} br
RewriteRule ^$ /.cache/index-fr.http2.dnt.html.br [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} gzip
RewriteRule ^$ /.cache/index-fr.http2.dnt.html.gz [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
```

```
RewriteCond %{HTTP:DNT} -eq1 [OR]
RewriteCond %{QUERY_STRING} X-ILSA-DNT=1
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteRule ^$ /.cache/index-fr.http2.dnt.html [L]

## homepage w/o DNT

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} br
RewriteRule ^$ /.cache/index-fr.http2.html.br [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteCond %{HTTP:Accept-Encoding} gzip
RewriteRule ^$ /.cache/index-fr.http2.html.gz [L]

RewriteCond %{SERVER_PROTOCOL} ^HTTP/2
RewriteCond %{HTTP_COOKIE} !SID1
RewriteCond %{HTTP_USER_AGENT} !^IlsaStaticRenderBot
RewriteRule ^$ /.cache/index-fr.http2.html [L]
```

*N.B.: The LZMA rules were removed from the examples above because they became obsolete when Safari started supporting Brotli.*

The concept in the rules above is that we want to serve static pages to:

- All new visitors (this is the SID1 cookie exemption);
- Except the robot that is in charge of building these pages (see the User-Agent rule).

Almost a decade ago, we used to build image sprites and to bundle CSS and JS files per feature in a bid to reduce costly network requests, and also to keep several domains for parallel downloads, but that hasn't been necessary for a while. At some point, HTTP/2 became so widely supported that we stopped maintaining HTTP/1.x optimisations.

Furthermore, the folder where these documents are stored can be mounted on a RAM disk for best performance.

Here is an example to adapt the rules for a more dynamic URL:

```
RewriteRule ^landing/sales/([a-z0-9_]+)(/[a-zA-Z]+)?$
/.cache/saleCodeRegister/$1-fr.http2.dnt.html.gz [L]
```

These Apache2 rules allow for high throughput of the pages that are cached in this way. Because of this, we have been able to withstand traffic generated through TV mentions and other publicity without the help of a CDN or any autoscaling.

## nginx

We have been using two nginx instances: one handles static assets while the other is a proxy in front of httpd and the other nginx instance.

The main idea is that we present the same URL for each asset to all browsers, regardless of their codec support and technology awareness. Server-side content negotiation finds the image best suited for each request. This makes it easy to cache the HTML documents for wide audiences, and also to reduce the amount of HTML for the `<img>` and `<picture>` tags.

### Serving text files

The basic idea with nginx is to enable the `gzip_static` and `brotli_static` directives so that compression variants are taken care of, and specialize your rewrite rules with `try_files`.

Our rules look for CSS and JS assets in the following order:

1. Minified;
2. Non-minified;
3. Original;
4. 404.

Here is an example of this:

```
location ~ ^/(css|js|static)/(.+)(?:\.\d+)\.(css|js) {
  gzip_static on;
  brotli_static on;
  try_files /$1/$2.min.$3 /$1/$2.$3 /$3/$2.$3 =404;
  add_header Cache-Control "public, immutable";
  expires max;
}
```

### Serving media files

The basic idea here is also the `try_files` directive but with all format variants. The directive will need to try expected formats (WebP, JPEG2000, *etc*.) and qualities (high resolution, normal resolution, possibly low resolution as well) one after the other.

Assuming a WebP compatible browser, the `try_files` directive will look for the images in the following order:

1. High resolution & WebP;
2. High resolution & JPEG;
3. Normal resolution & WebP;
4. Normal resolution & JPEG;
5. 404.

First, we need to `map` a few variables in the main `http` block. This sets the following variables so we can use them from anywhere in the config files: `$webp_suffix`, `$jp2_suffix`, `$jxr_suffix` and `$dpr`.

```
# WebP
map $http_accept $webp_suffix {
  ~image/webp  ".webp";
  default   "";
}

# JPEG-2000
map $http_user_agent $jp2_suffix {
  "~iOS.+Chrome/[0-9.]+ Safari/[0-9.]+$" ".jp2";
  "~Chrome/[0-9.]+ Safari/[0-9.]+$" "";
  "~Safari/[0-9.]+$" ".jp2";
  default   "";
}

# JPEG-XR
map $http_user_agent $jxr_suffix {
  "~Edge/[0-9.]+$" ".jxr";
  default   "";
}

# Device Pixel Ratio
map $http_ch $dpr {
  "~*dpr=(?<value>[0-9.]+)" $value;
  default   "1";
}
```

Now set up a few other variables in each `server` block and compose all of them into the `$dpr_suffix` and `$format_suffix` variables, which will be required for the `try_files` directive.

```
set $origin_domain "www.instantluxe";

set $dpr_suffix "@${dpr}x";
if ($dpr_suffix = "@1x") {
  set $dpr_suffix "";
}

set $format_suffix ".jpg";
if ($jxr_suffix) {
  set $format_suffix $jxr_suffix;
}

if ($webp_suffix) {
  # override any other image encoding, this one is usually better
  set $format_suffix $webp_suffix;
}

if ($http_user_agent ~ "iPad|iPhone|Macintosh|Intel Mac OS X|macOS|iOS|Android") {
```

```
  set $dpr_suffix "@2x";
}
```

Finally, we can prepare the `location` blocks with the actual `try_files` directives.

For reference, here is the structure of the main photo URL:
`/category/subcategory/brand/keywords-ref-size.version.jpg`

Example:
`https://assets.instantluxe.com/maroquinerie/sac/chanel/sac-chanel-mademoiselle-noir-cuir-femme-A208808-1600.1.jpg`

Nginx config:
```
# Product main photo
location ~
^(?<path>/[a-z0-9/-]+)-A(?<prefix1>\d)(?<prefix2>\d)(?<suffix>\d+)-(?<size>\d+)(?:\.\d+)?\.
jpg$ {
  root /home/storage/media;
  set $local_filename
/thumbnails/product/$prefix1/$prefix2/$prefix1$prefix2$suffix/main/$size;
  try_files $local_filename$dpr_suffix.jpg$format_suffix $local_filename$dpr_suffix.jpg
$local_filename.jpg$format_suffix $local_filename.jpg =404;
  add_header Cache-Control "no-transform, public, immutable";
  expires max;
}
```

Secondary images (gallery):
`/category/subcategory/brand/keywords-ref-size-position.version.jpg`

Example:
`https://assets.instantluxe.com/maroquinerie/sac/chanel/sac-chanel-mademoiselle-noir-cuir-femme-A208808-1600-a.1.jpg`

Nginx config:
```
# Product gallery photo
location ~
^(?<path>/[a-z0-9/-]+)-A(?<prefix1>\d)(?<prefix2>\d)(?<suffix>\d+)-(?<size>\d+)-(?<letter>[
a-z])(?:\.\d+)?\.jpg$ {
  root /home/storage/media;
  set $local_filename
/thumbnails/product/$prefix1/$prefix2/$prefix1$prefix2$suffix/gallery/$letter/$size;
  try_files $local_filename$dpr_suffix.jpg$format_suffix $local_filename$dpr_suffix.jpg
$local_filename.jpg$format_suffix $local_filename.jpg =404;
  add_header Cache-Control "no-transform, public, immutable";
  expires max;
}
```

For example, say we are requesting the 490px size of the main photo mentioned above:
`https://assets.instantluxe.com/maroquinerie/sac/chanel/sac-chanel-mademoiselle-noir-cuir-femme-A208808-490.jpg`

Here are the variants on disk:

```
-rw-rw-r-- 1 instantluxe www-data 74829 2018-10-12 15:40 490@2x.jpg
-rw-rw-r-- 1 instantluxe www-data 68942 2018-10-12 15:40 490@2x.jpg.webp
-rw-rw-r-- 1 instantluxe www-data 23571 2018-10-12 15:40 490.jpg
-rw-rw-r-- 1 instantluxe www-data 20854 2018-10-12 15:40 490.jpg.webp
```

An iPhone would receive a `content-type` of "image/jpeg" with a size of 74 KB, while a typical desktop Chrome would receive a `content-type` of "image/webp" with a size of 20 KB.

## Benchmarking the homepage

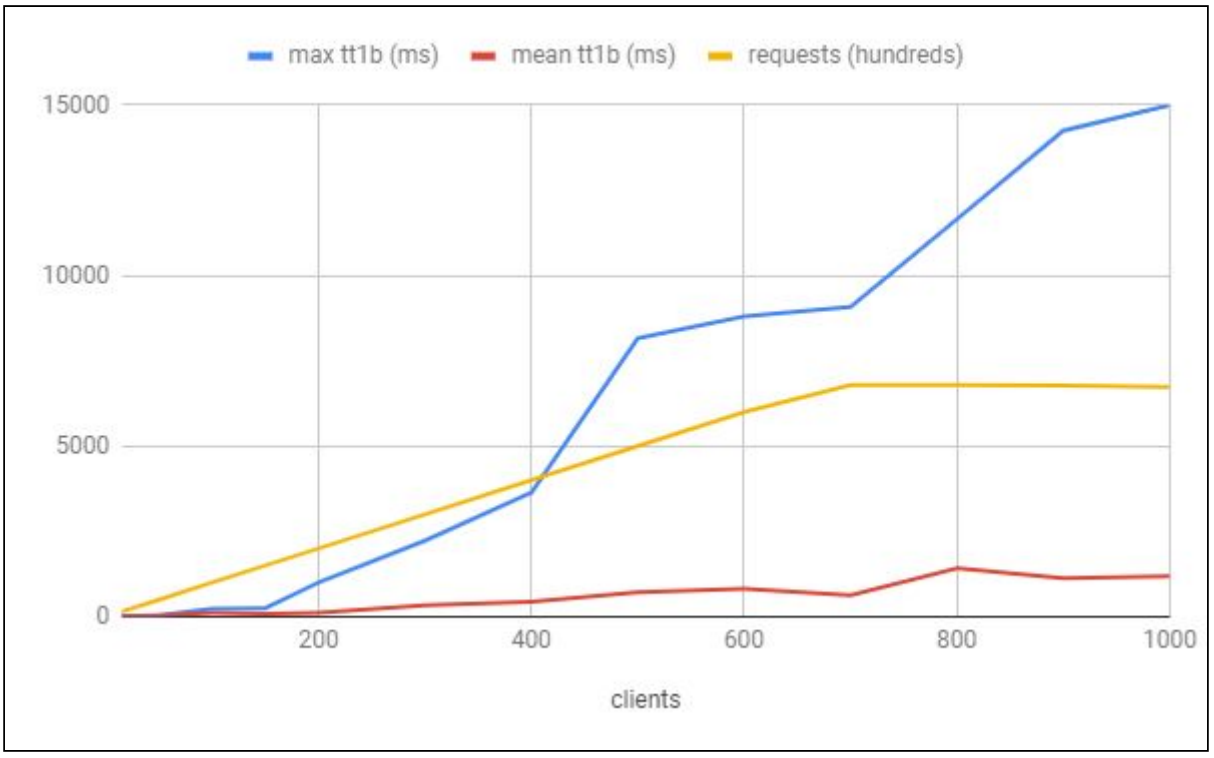
We can use `h2load` and change only the number of clients over the same 60 second period, using the following as a template (here with 200 clients):

```
$ h2load -D 60 -c 200 -t 10 -H "Accept-Encoding: br" https://www.instantluxe.com
```

Obvious disclaimer: this is not representative of actual users, firstly because h2load doesn't parse HTML and therefore it doesn't download any resources, and also because real users don't come in a steady stream or all at once. It means that what we are benchmarking here is a worst-case scenario.

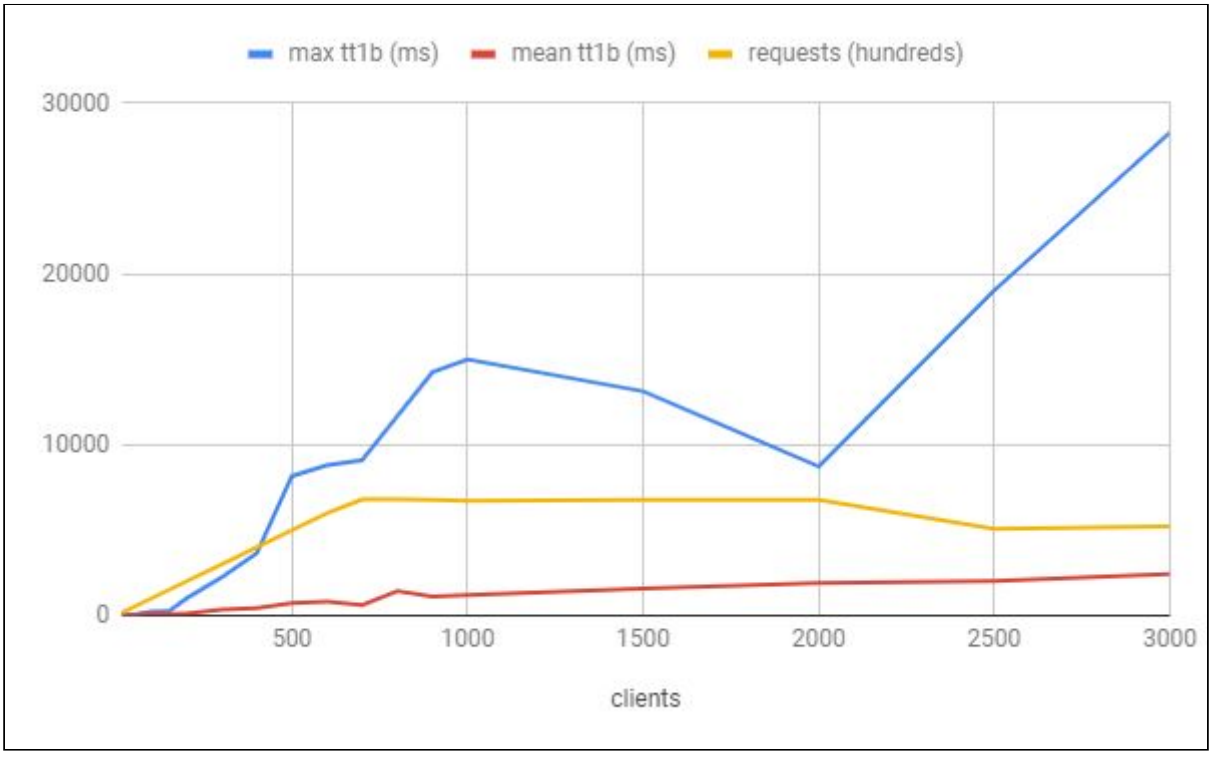
The following graph represents the speed at which the web server can deliver the homepage over the network, as well as the resilience to a sudden influx of visits. The metrics shown are the number of requests per second that were sent, the total volume of received data, the max time to first byte and the mean time to first byte.

*NB: to get the number of requests to fit in the same graph, they are represented as hundreds. For example, at 700 clients and above, the number of requests caps at just below 700,000.*




*Homepage throughput benchmark (0-1000 simultaneous clients)*

Up to 700 clients, the time to first byte is consistently below 1 second for all requests. After that point, the testing tool started receiving responses above that threshold (and increasing).




*Homepage throughput benchmark (0-3000 simultaneous clients)*

The latency decreases between 1000 and 2000 clients, probably due to the way nginx creates worker processes to handle an increasing load. Above 2000 clients, some requests start taking longer, affecting the number of received responses while the mean time to first byte increases ever so slightly.

While these tests (millions of requests) were running on the Production infrastructure, the origin web server didn't get more than a few hits, all of which were served from disk; the database also didn't get a single query from this benchmark.

Thanks to this setup, our company never had a campaign that would strain the infrastructure. Indeed, our niche market meant that this proxy setup was always powerful enough to handle the load. Further improvements would likely involve a CDN service.

## Network latency

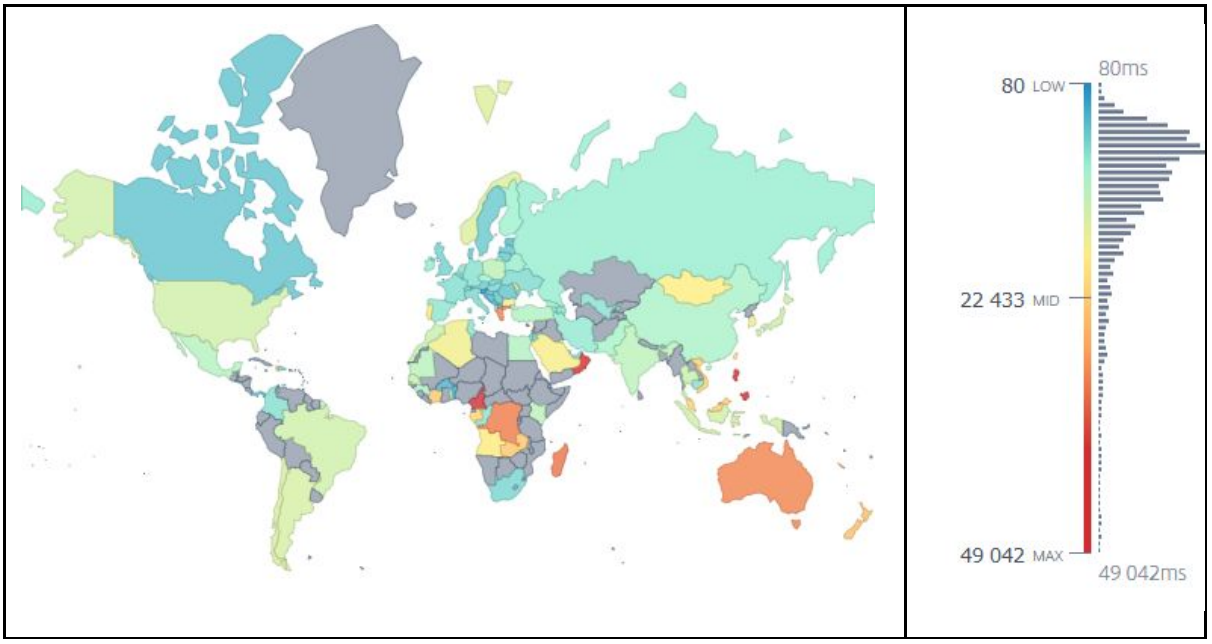
We have used the services of a CDN a few times.

However, since our customers are 63% from France, 9% from Italy and generally 80% from Europe (not counting Russia), the CDN service would be useful to less than 20% of our customers. Because we ship goods of high value, there are import/export taxes to consider and their impact on expected transaction rates and customer satisfaction; in this context, network latency is less of an issue for the remaining 20%.

When we gave CDN services a try, there were still regions that had high latency while we would have liked those regions (China) to be significantly faster than without the CDN. We didn't have the time to monitor this closely or to set up intelligent CDN failover based on real-time data.

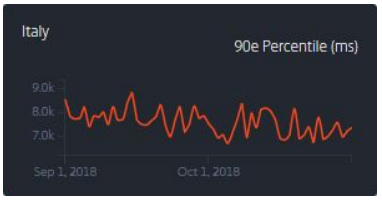
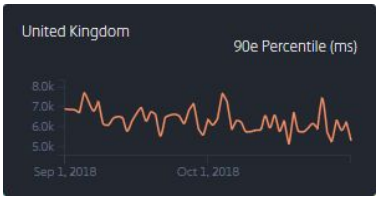
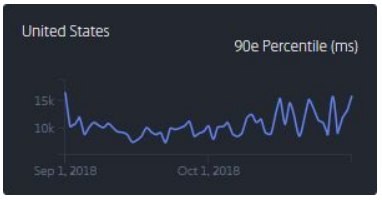
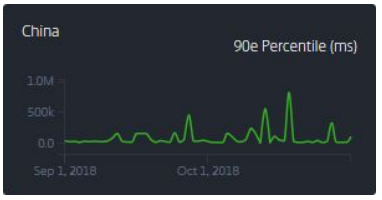
So, we trusted our hosting provider to work on their peering with other networks. Whenever we knew that anything was amiss, we got in touch with them so they could investigate.

Here are some latency numbers to get an idea:




*InstantLuxe.com's "total page load time" according to Cedexis (last 30 days on Nov. 22nd, 90th percentile)*

Some of the countries that were the most interesting to us:

| | | |
|---|---|---|
|   France : AVG 4s, P90 7.7s |   Italy: AVG 4s, P90 7.5s |   UK: AVG 3.5s, P90 6.2s |
|   Belgium: AVG 3.7s, P90 6.7s |   US: AVG 5.8s, P90 10.6s |   |

*InstantLuxe.com's "total page load time" per country according to Cedexis (last 30 days on Nov. 22nd)*

# Scripts

We find that the closest they are to the machine, the better we can compose our tools. Therefore, we usually install binaries from well-known open source initiatives and we build Bash (Linux) scripts to wrap and to chain them.

Why Bash? It is exceptionally well documented and it is available natively on all major platforms: Linux and macOS, and now Windows as well if you install the Windows Subsystem for Linux[16].

For tasks where Bash is ill-suited, we have been using another language. In our case, this is PHP but you should definitely keep your favorite.

There are going to be several categories of scripts: the launchers, the compressors, the encoders and the one that validates the improved files.

# Setting up

*Disclaimer: this chapter is **not** a complete server setup, and definitely not a security best-practices guide. This is not even a minimum viable production setup. We will merely give some performance pointers.*

## Using a temporary virtual machine for the tests

Testing with VirtualBox: Make sure the network card is configured as “Virtual private host”. This will add an IP to your local machine on the same subnet as what VirtualBox uses (for example 192.168.56.0), so you will be able to call the virtual machine from your own machine.

You should also allow SSH from the virtual machine’s terminal in the following file:
`vi /etc/ssh/sshd_config`
Find the PermitRootLogin option, uncomment it if necessary and set its value to “yes”:
`PermitRootLogin yes`
Save and close the file, then run “`service ssh reload`”.
Finally, install the net-tools package so we can find out the VM’s IP address:
`apt-get update && apt-get upgrade && apt-get install net-tools && ifconfig`
You can now run ssh into your virtual machine from your own machine (change the IP below by the one that was given by `ifconfig` above):
`ssh root@192.168.56.101`

If everything works, you should save a snapshot of the VM. From now on, you can choose the client to connect to the VM, and you are not constrained by VirtualBox’s limited console.

## Server performance tips

### Use the RAM to store temporary and cache files

*Disclaimer: this is actually a production-ready tip, but do experiment first.*

[16] https://docs.microsoft.com/en-us/windows/wsl/install-win10

We are going to write a lot of temporary files to disk. To improve performance for this kind of input-output (i/o), we are going to use a RAMdisk instead of the actual hard drive or SSD. This way, the server will write the many temporary PNG and JPEG files to RAM and it will be a lot faster. You can check the busy state of your disks with the "`atop`" tool.

For those who need convincing beforehand, you could test the speed at which your system can write files. Using an 1.5 G file as an example, copying it to an SSD takes up to ~20s on our system; on the other hand, copying the same file to a RAMDisk takes ~1s.

This tip won't help speeding up the actual encoding of the images, because that typically requires CPU time this tip doesn't address. The encoding can sometimes be parallelized over several CPU, and some tools have options for that (see "threading"), but in that case you would need more RAM and/or CPU cores to handle the same tasks, and that's not in the scope for this article. Just remember that each tool's contributors are aware of their program's CPU and RAM usage and they are doing their best to make them as efficient as possible.

The RAMdisk tip is very useful if your drives are SSD, because writes wear them down and we are going to do a lot of writes. It is also going to be just as useful for hard disk drives because they are very slow with writes, while RAM is fast. Either way, it's a win.

Since we want to refrain from using the `root` user for everything, we create a user that we will call "**`ilsa`**" from now on. All tasks will be run by that user, and the temporary folder will be theirs too.

```
su ilsa
mkdir /home/ilsa/tmp
```

Now add something like this to the `/etc/fstab` file:

```
tmpfs   /home/ilsa/tmp   tmpfs
nodev,nosuid,noexec,nodiratime,noatime,size=8G   0 0
```

Beware the "8G" in this example, it's the amount of space that this mount point will be allowed to grow to. Use a low number and your encoding scripts will fail; use a number too high and you will deplete your RAM. Don't use any of this *verbatim*, but read the docs and adapt to your use case.

Now run "`sudo mount -a`" to apply the changes and "`df -h`" to check them:

```
Filesystem      Size  Used Avail Use% Mounted on
[...]
tmpfs           8.0G     0  8.0G   0% /home/ilsa/tmp
```

If you have a write-heavy tool at hand, you can run `atop` in a console and watch as it behaves differently when you use the disk or the new mount point.

Don't forget to periodically clean up the new directory, or you will fill the space really fast. This can be done with a crontab, here's an example:

```
* * * * * /usr/bin/find /home/ilsa/tmp -ignore_readdir_race -mmin +15 \! -type d -delete
```
*(delete all non-directory files written at least 15 minutes ago)*

### Use a functional approach for scripts

As programmers, some of us are used to create lists we can iterate over. With Bash and especially when handling files, this is not always the best approach. Bash can do enumerations and loops, but it can also chain commands, the output from one becoming the input to the next (with very efficient streams between them). Or at least, you should try to loop over streams rather than strings and lists.

In this, Bash is closer to SQL than to imperative languages.

## A word on “find”

You may find that this tool is very powerful. It allows for criteria to locate files on the filesystem, so we are going to use it a lot.

One nice bonus with “find” is its action parameter, which you can set to “print” while you are testing your filters and then change to “delete” once you are ready, or you can use “exec” and have it run another command with parameters you specify.

Here’s how to find old temporary files and do something with them:
```
find /home/ilsa/tmp -ignore_readdir_race -mmin +15 \! -type d -print
find /home/ilsa/tmp -ignore_readdir_race -mmin +15 \! -type d -delete
find /home/ilsa/tmp -ignore_readdir_race -mmin +15 \! -type d -exec ls -alh '{}' \;
```

Other examples are:

Non-directory files that were written at least 5 days ago, in absolute days
```
find /home/ilsa/tmp -daystart -mtime +5 \! -type d -print
```

Retina-ready JPEG files
```
find /home/ilsa/tmp -name '*@2x.jpg*' -print
```

Empty files with a recompressed suffix
```
find /home/ilsa/tmp \
  -regextype posix-egrep -regex '.+\.(gz|br|lzma|webp|jp2|jxr)$' \
  -size 0 \
  -print
```

Pass GZ and Brotli filenames to a custom script with a command-line parameter of “2”
```
find /home/ilsa/tmp
  -regextype posix-egrep -regex '.+\.(gz|br)$' \
  -exec /home/ilsa/scripts/obsolete-alts.sh 2 '{}' \;
```

Or if you dislike this syntax, you can use the “print” action and pipe `find`’s output to the other program
```
find /home/ilsa/tmp -regextype posix-egrep -regex '.+\.(gz|br)$' -print \
  | /home/ilsa/scripts/obsolete-alts.sh 2
```

## Installing the binaries

You can choose to install them from any package manager of your choice, or you can build them from source. We won't go into the details of installing each here.

Here are the dependencies we are going to review:

Brotli[17]
https://github.com/google/brotli
Installed as:
`/usr/local/sbin/brotli`

LZMA (for Safari, rendered obsolete by Brotli)
https://tukaani.org/xz/
Installed as:
`/usr/local/bin/lzma`

JPEG-XR[18] (for Edge, rendered obsolete by WebP[19])
https://jxrlib.codeplex.com/releases/view/107208
Installed as:
`/usr/local/bin/JxrEncApp`
Sometimes, it is already installed with the system. Use "`which JxrEncApp`" to find out if this is the case.

Zopfli (supported by all browsers)
https://github.com/google/zopfli
Installed as:
`/usr/local/sbin/zopfli`

WebP[20] (for the longest time it was supported only by Chrome, now also Edge and Firefox)
https://github.com/webmproject/libwebp
Installed as:
`/usr/local/bin/cwebp`
`/usr/local/bin/gif2webp`

ImageMagick
https://github.com/ImageMagick/ImageMagick/releases
Installed as:
`/usr/local/bin/magick`

JPEG-2000 from OpenJPEG

[17] https://caniuse.com/#feat=brotli
[18] https://caniuse.com/#feat=jpegxr
[19] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/webpimageformat/
[20] https://caniuse.com/#search=webp

https://github.com/uclouvain/openjpeg/
Installed as:
`/usr/local/bin/opj_compress`

Guetzli
https://github.com/google/guetzli
Installed as:
`/usr/local/bin/guetzli`

Butteraugli
https://github.com/google/butteraugli
Installed as:
`/usr/local/bin/compare_pngs`

There are many other tools (notably **ImageOptim**), as this space is evolving fast. We haven't tried them all. Some names appeared recently. We aren't a CDN, we are just trying to provide a decent website for our customers. There is a point after which more engineering work won't help the bottom line.

## About the perceptual difference with image encoding

There are several ways to compute the perceived difference of two images, mostly to ascertain which of two compression ratio looks best to the human eye. Butteraugli is one of them and it calls this "psychovisual difference".

There are several tools around the same issue and we will try to give an overview of how they differ.

Butteraugli: compare two existing versions of the same PNG image
It won't work on images that are essentially different, and it will also complain if the dimensions are different. It is intended to identify the best compression ratio for one image, by iterating over multiple compression settings.
Be aware that it is not immediate; it may take several dozen seconds for the analysis to complete.

Guetzli: a JPEG compressor
To best use this tool, you have to know in advance the best JPEG quality setting for your use case or let the default. Guetzli won't find the best quality setting for you, but rather the best image for the quality setting you are targeting. It works by applying several quantization techniques (that's one of the stages to encode a JPEG) for one quality setting and applying butteraugli's techniques to compare the results so as to produce only the image with the best score.
Be aware that it is very **slow** (several minutes for each file).

## Composing the toolchain

As stated before, Linux and Bash are very efficient at chaining commands. The buffer works well to parallelise commands up to a point, but we are talking about binary files here. We shouldn't pass around so this type of content between scripts, mostly because Linux pipes are made for text, which means they are used to split input by using the newline characters, and that doesn't work well with binary content. Therefore, when chaining compression tasks, we will pass around file names rather than their contents.

The easiest way to batch-process images is to echo lists of filenames to the standard output that we pipe into the next script, which in turn echoes the filenames so they can be piped further. This way, we gain a measure of parallelism for free thanks to the very efficient streaming of the pipes (the calling script will pause when its buffer is full, and resume when it gets free).

To that end, we will need a small Bash wrapper for each tool so we can standardize their usage. We will call our wrappers brotlify.sh, zopflify.sh, webpfy.sh, jp2fy.sh *etc*.

Here is an example of the proposed toolchain:

```
find /path/to/images [options] -print | webpfy.sh | jp2fy.sh
find /path/to/assets [options] -print | gzipfy.sh | brotlify.sh | zopflify.sh
```

Each wrapper will have the same logic:

1. try to create an optimized file in a temporary location;
2. keep the new file only if its size is smaller than the original;
3. if a file is already present where the optimisation was going to be placed, compare their hash before renaming the file;
4. if everything checks out, update the new file metadata from the original and place the new file at the final location.

Individually, some wrappers will require additional steps. For example, the JP2 converter only accepts PNG files as input, which means that for non-PNG files we will need to convert them on the fly.

We could make each tool echo only the filenames it processed but that would break the chain, so instead we will have each wrapper echo all the filenames it sees.

## About temporary files

To perform our optimisations, we need to create alternate compressed files for each of the files that need to be published. On average, we will try 4 encoders per image and 3 encoders per text file, without knowing in advance if any of the resulting files will be good enough to keep.

If we create a JP2 that is bigger than say the JPEG it came from, we don't want that file to reach our users. We need to be ready to drop the JP2 file without it ever reaching the final location on the disk. We don't want to disturb the final files on disk unless they need to be changed.

This means we need a way to create temporary process files so that they don't disturb the final filenames until they are validated.

It looks like this:

1. Create the compressed version of a file at a temporary location;
2. Compare its file size with the file it was created from, and drop it if too big;
3. Place the alternate compressed file at its final location, with metadata already set.

## Wrappers

For ease of use and ease of chaining, every compression format will have its own wrapper script.

The most simple wrapper could look like this:

```
#!/usr/bin/env bash
{
COMMIT_BIN="/usr/local/bin/commit-file.sh"
WEBP_BIN="/usr/local/bin/cwebp"
WEBP_OPTS="-alpha_cleanup -quiet -m 6 -q 90 -sharpness 0 -metadata none -mt"
EXT="webp"
TMP_PATH="/path/to/tmp"

while read SRC_FILENAME
do
  if [ -s "$SRC_FILENAME" ]
  then
    echo "$SRC_FILENAME"

    TMPFILE=$(mktemp $TMP_PATH/XXXXXX.$EXT)
    $WEBP_BIN $WEBP_OPTS "$SRC_FILENAME" -o "$TMPFILE"

    $COMMIT_BIN "$SRC_FILENAME" "$TMPFILE" "$SRC_FILENAME.$EXT"
    rm $TMPFILE
  fi
done

exit
}
```

Here is a walkthrough:

The curly braces and the "`exit`" keyword enclosing the script are there to make sure that if we update the script while it is running, the current run won't break unexpectedly. Indeed, if you modify a script while it is running, your modifications will be taken into account by the already-started process. Say you add a block of code at line 18 while it was already running

at line 20: when it reaches the end of that line, it will read line 21 which is not what it was when it started, and hopefully it will cause a syntax error, or in the worst case it won't immediately fail but start running code that was not meant to be run. So, depending on the modifications, it may be very bad. Curly braces avoid that by forcing Bash to read the whole program in memory as soon as the script starts.

First we set a few variables. Then we loop over the standard input, which means every line piped to this script will cause one iteration of the loop to be read into the `$SRC_FILENAME` variable.

The second action is to encode the original file with WebP at the temporary location, and finally we call the common "commit" script which will check a few things (permissions, file size, timestamps *etc*.) before copying the temporary file to its final location. This line is where we check that the new file has a lower file size than the original and then does basically a "`cp`" command that makes sure all attributes are copied from the original file to the final file, whatever the attributes of the temporary file.

For the target filename, the naming convention we came up with was to simply add a suffix to the original filename.

## The launcher script

Here is a proposed launcher script, suitable for a cronjob that could run every minute:

```
#!/usr/bin/env bash
{
EMAIL="admin@example.com"
FIND_GIFS="-regextype posix-egrep -regex '.+\.gif'"
FIND_IMGS="-regextype posix-egrep -regex '.+\.(jpg|png)'"
FIND_OPTS="-prune -writable -mmin -2"
WEBP="/usr/local/bin/webpfy.sh"
GIF2WEBP="/usr/local/bin/gif2webpfy.sh"
JPEG2000="/usr/local/bin/jp2fy.sh"
JXR="/usr/local/bin/jxrfy.sh"
TMP_PATH="/path/to/tmp"

function finish () {
  if [ -e "$ERRORLOG" ]; then
    rm $ERRORLOG
  fi
}

trap finish EXIT

# temporary files
ERRORLOG=$(mktemp "$TMP_PATH/XXXXXX.log")

# find GIF images and pipe their filenames to a script
find /path/to/images $FIND_GIFS $FIND_OPTS -print \
  | $GIF2WEBP 2>$ERRORLOG

# find JPG and PNG images and pipe their filenames to a script
find /path/to/images $FIND_IMGS $FIND_OPTS -print \
  | $WEBP 2>$ERRORLOG \
  | $JPEG2000 2>$ERRORLOG \
  | $JXR 2>$ERRORLOG

# filter out useless (non actionable) errors from the log
cat $ERRORLOG 2>/dev/null | grep -v \
  -e "file size changed while zipping" \
  -e "chown failed: Operation not permitted" \
  -e "mv: cannot create regular file" \
  -e "Invalid filename:" \
  >$ERRORLOG

# in case of any error, do something (here we send them over email)
if [ -s $ERRORLOG ]
then
  mail -s "job errors: re-encode images" $EMAIL <$ERRORLOG
  exit 1
fi

exit 0
}
```

## Committing the new file

Below is an example script to copy the temporary file at its new location:

```
#!/usr/bin/env bash
{
SRC_FILENAME="$1"
TMPFILE="$2"
DST_FILENAME="$3"

if [ ! -s "$SRC_FILENAME" ]
then
    echo "$SRC_FILENAME" >/dev/null
    exit 0
fi

if [ ! -s "$TMPFILE" ]
then
    echo "$TMPFILE" >/dev/null
    exit 0
fi

#>&2 echo "committing $SRC_FILENAME to $DST_FILENAME via $TMPFILE..."
if [ ! -f "$DST_FILENAME" ]
then
    touch "$DST_FILENAME"
fi

SRC_SIZE=$(stat -c%s "$SRC_FILENAME")
DST_SIZE=$(stat -c%s "$TMPFILE")
EXIST_SIZE=$(stat -c%s "$DST_FILENAME")

#>&2 echo "orig=$SRC_FILENAME=$SRC_SIZE B | work=$TMPFILE=$DST_SIZE B |
live=$DST_FILENAME=$EXIST_SIZE B"
if [ "$DST_SIZE" -gt "$SRC_SIZE" ]
then
    #>&2 echo "skipped $TMPFILE as $DST_FILENAME (bigger than the original)"
    rm "$DST_FILENAME"
else
    # note: we aren't copying directly at the final location
    # because the copy may take time and requests are coming in
    # 1- copy all attributes from the original
    touch --reference="$SRC_FILENAME" "$TMPFILE" 2>/dev/null
    chmod --reference="$SRC_FILENAME" "$TMPFILE" 2>/dev/null
    chown --reference="$SRC_FILENAME" "$TMPFILE" 2>/dev/null
    # 2- move the new file to its final location
    mv "$TMPFILE" "$DST_FILENAME"
    #>&2 echo "copied $TMPFILE to $DST_FILENAME"
fi

exit
}
```

## Putting it all together

Set the launcher script as a cronjob as often as you like, depending on the time constraints you applied to the `find` command in your scripts. You want the job to run at least as often as these constraints will be looking for files.

If you plan to run the script often, you may want to to ensure that the same images aren't processed several times. This can be done with lock files dropped in the /var/run folder. For example:

- Compute the hash of the input file name and use it as the file name of a lock file;
- If the lock file doesn't exist, write the current PID to the lock file;
- Otherwise, check its timestamp to abort the previous run or the current run of the encoder script and then update the lock file if necessary.

## Improving even further

When an original file is updated on disk, we can be reasonably sure that any alternate formats will be obsolete. We don't need to (and we shouldn't) wait for the encoding to finish to determine this, and we should delete these files as soon as the process is started. This could be a good first step in each chain of encodings: delete all alternate formats whose timestamp are older than the original file they were made from.

Another area for improvement would be to watch for file updates (possibly using `inotifywait`) instead of scheduling the process, so that the encoding can start as early as possible.

Finally, the chain of encodings could be done in parallel instead of sequentially, since each format is not dependent on the result of any other format.

However, you may want to watch out for the number of processes running at the same time. You may use up all the memory or even reach the maximum number of processes your system can handle at once. At that point, you may want to look into splitting your batch processing load.