29 NOVEMBER 2018 / #CLOUD COMPUTING

Distributed Systems: When you should build them, and how to scale. A step-by-step guide.

Emmanuel Marboeuf

Read more posts by this author.

It always strikes me how many junior developers are suffering from **impostor syndrome** when they began creating their product.

I get it, there are many **mind-blowing** examples of **top companies** with incredibly complex distributed systems that can tackle **billions of requests**, gracefully upgrade hundreds of applications without any downtime, recover from disaster in seconds, release every 60 minutes, and have light speed response times from anywhere in the world.

These expectations can be pretty **overwhelming** when you are starting your project. But as many of you already know, a majority of these companies have started with a **minimal viable system** and a **very poor technology stack**. There is a simple reason for that: they **didn't need it** when they started. Spending more time designing your system instead of coding could in fact cause you to **fail**.

This article is a **step by step** how to guide. I will show you how, at Visage, we started with the tiniest system ever and built a basic high availability scalable distributed system. This is a real **case study** to **remove** your **complexes** if you have never had the opportunity to do it yourself.

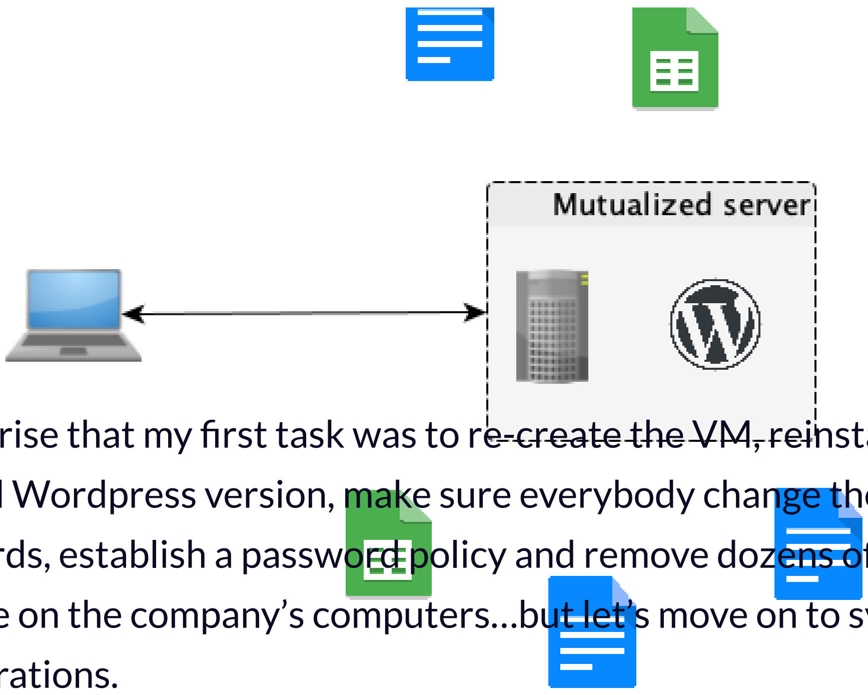
When I first arrived at Visage as the CTO, I was the only engineer. I knew nothing about the tech stack, but I joined because I really liked the idea of being able to **recruit without** in-house **recruiters** or an HR service. This was the core idea behind Visage: **crowdsourcing** powered by a lot of invisible recruiters working together on your roles assisted by **artificial intelligence** that would look for the most suitable no middle man.

The “**crowd**” in crowdsourcing instantly triggered my engineering brain: there are going be a **lot of people**, working **concurrently**, expecting **good performance** from **anywhere** in the world. I liked the challenge.

But system wise, things were **bad**, **real bad**. This is what I found when I arrived:

- A compromised Wordpress instance running hundreds of outdated flawed plugins, running in a VM on a shared server
- Compromised mailboxes
- A crap ton of Google Docs and Spreadsheets.

And **this is perfectly normal.** Again, there was no technical member on the team, and I had been expecting something like this. Still the team had focused on a business opportunity and made the product seem like **it worked** magically while doing everything manually! (Fake it until you make it). And that’s what was really amazing.



No surprise that my first task was to re-create the VM, reinstall an updated Wordpress version, make sure everybody change their passwords, establish a password policy and remove dozens of malware on the company's computers…but let's move on to systems considerations.

Our first system (Yes, it sucked but it did the job) !

From Wordpress to a web application

Your first focus when you start building a product has to be **data**. Data is what drives your **company's value**. It will be what you use everyday to make decisions, and what you show to your **investors** to demonstrate **progress**.

You need to make sense of your data, and recouping your data from different sources with different formats is gonna be a **huge waste of time**. Wordpress can be a very good choice in many cases by saving quite a lot of engineering time, but for their needs, the Visage team had to install fancy plugins that were not maintained anymore. As a result we had no control over the generated data model, and data that couldn't fit the model was scattered across dozens of docs and spreadsheets.

So unless there is a product out there that already fits **90%** of your needs, think about an **ideal data model** and **design** and **implement** a

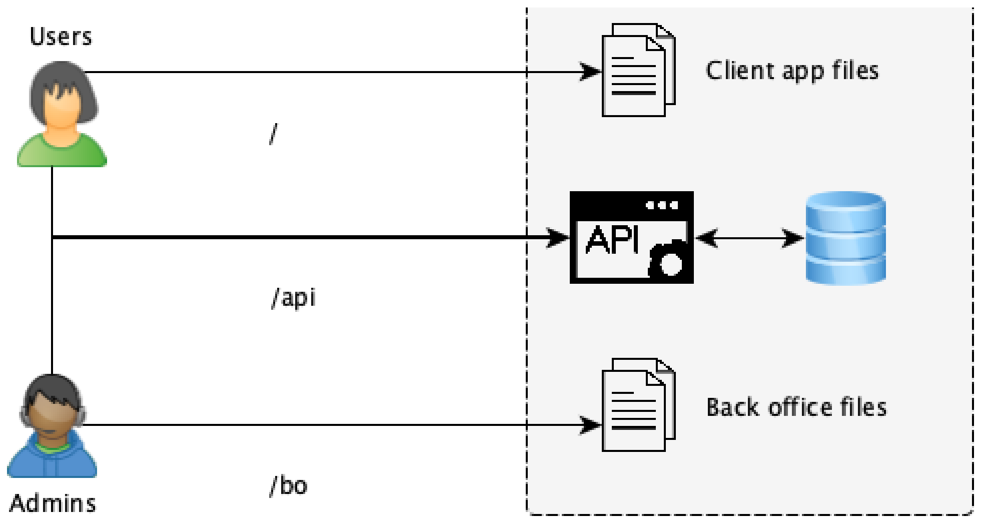
data.
Then think **API**. Your application must have an API, it's going to be critical when you eventually sell it. Don't immediately scale up, but code with scalability in mind. Make your API **stateless** and as **RESTful** as you possibly can since everybody will expect to be able to query it using standard HTTP methods.

We chose **NodeJS** in our case, because most of our code would just be processing inputs and outputs. NodeJS is **non blocking** and comes with a library that is convenient to design APIs: **ExpressJS**.

If you need a customer facing website, you have several options. First you can create a layer in your application server that will generate your pages or you can build a **Single Page Javascript** application that will be served by a static web hosting server.

At Visage, we went for the second option and decided to create one application for **users** and one for **admins.** This was simply because we would have much bigger expectations for users than we needed with admins, and wanted to keep both codebases **simple** (also, for **CORS** considerations later on). This is what our system looked like:

Donate



All data in one place

Delegate sensitive data storage early on

Unless it's critical to your business, there is no good reason to store sensitive personal data in your systems. Security is a complex matter, and if you are modifying your code everyday until you find your product market fit, it will break. Assume that **anybody ill-intended could breach your application** if they really wanted to.

The key here is to not hold any data that would be a quick win for a hacker. **Nobody robs a bank that has no money**. If you are designing a SaaS product, you probably need authentication and online payment. There are a lot of **third parties** you can integrate with that will deal with that in a much better way than you possibly could .

Auth0, for example, is the most well known third party to handle Authentication. Stripe is also a good option for online payments. They will dedicate **all** their resources and the best **security** engineering business.

Actual sign on a car in San Francisco

Cloud services are your best friends

So at this point we had a way to store all our data, authentication, online payment, and a web app that clients could use along with an API that we could sell to partners for different use cases. Our user base was growing and it became obvious that they wanted to be able to access the app anytime. So it was time to think about **scalability** and **availability**.

We were relying on one server but it could only handle so many requests, and changing servers or releasing a new version would mean were: **load-balancing**, **auto-scaling**, **logging**, **replication** and **automated back-ups**. Of course, if you are the only engineer in your company, trying to tackle all these issues on your own would be complete madness.

Luckily we live in a time that just a single well rounded engineer can easily build such a system in a couple of days using Cloud services like **Amazon Web Services**, **Google Cloud Services** or **Azure**. We decided to move our systems to **AWS** because at that time it was the most complete solution and we had 2 years of free credits.

This is why I am mostly gonna talk about AWS solutions in this post, but there are equivalent services in other platforms. This is also the time we chose to start running our modules in **Docker containers** for a lot of different other reasons that will not be covered in this post (you can check out this article for more info: https://medium.freecodecamp.org/amazon-fargate-goodbye-infrastructure-3b66c7e3e413).

How you decide to run your applications really depends on your use-case, like the **flexibility** you need versus **the time** you can spend managing your infrastructure.

There is no good or bad answer.

You can choose to containerize all your modules and use a **container management system** like ECS/EKS in AWS or Kubernetes engine in GCP. If not and you don't want to deal with things like auto-scaling and load-balancing yourself, you can use Elastic Beanstalk or App Engine.

If you want to go full **Serverless** you can also combine the use of Lambda functions and API Gateway. We decided to go for **ECS.** We up **auto-scaling** depending on CPU usage, integrated all our containers' logs with **Cloudwatch** and set-up Metrics to watch **errors**, **external calls** and **API response time**.

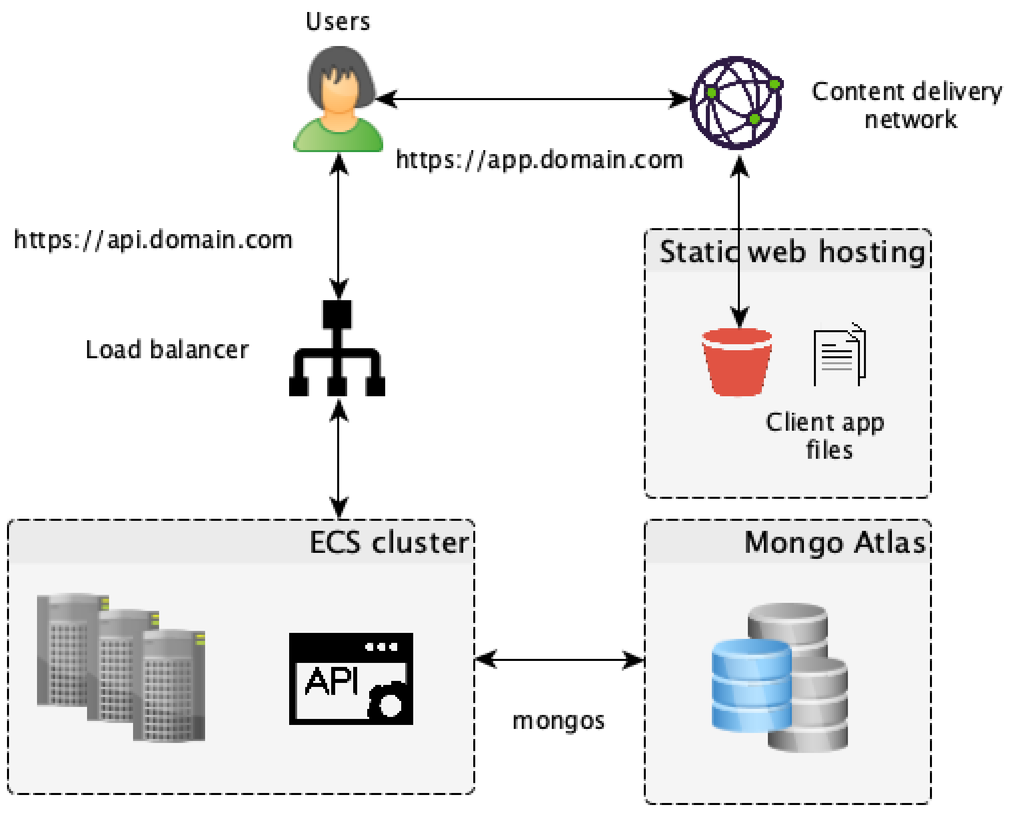
High Availability: Did you know that Giraffes almost never sleep ? **99% uptime**

For our Database, we used MongoDB, because our model is a good fit for a NoSQL database, and for its high consistency. We decided to take advantage of **MongoDB Atlas** and deployed 3 replicas to allow for high availability. Among other services, Atlas provides **auto-scaling**, **automated back-ups** and allows you to go **back in time** seamlessly in case of disaster.

We also decided to host all our static web files in **S3** and used **Cloudfront** as a **CDN** so our JS apps can load very quickly anywhere in the world and be served as many times as requested. **Cloudfare** is also a good option and offers a DDOS protection out of the box.

name servers for all our domains. This is one of my favorite services on AWS. It makes your life so much easier. Every time you want to serve something through a domain name, whether it's an **EC2** instance, an **elastic IP**, a **load-balancer**, a **Cloudfront distribution** or anything really, privately or publicly, it takes you minutes because it's so well integrated with all the other services.

Combine that with the **Certificate Manager** that allows you to get **SSL certificates** (wildcards included) for free in minutes and to deploy them on all your servers by ticking a box, and you have the fastest most reliable way to enable **HTTPS** on all your modules. Good bye "Let's Encrypt" SSL certificates that I had to renew and install on my servers every 3 months or so ?.



Starting to look decent

Donate

Decide on a caching strategy

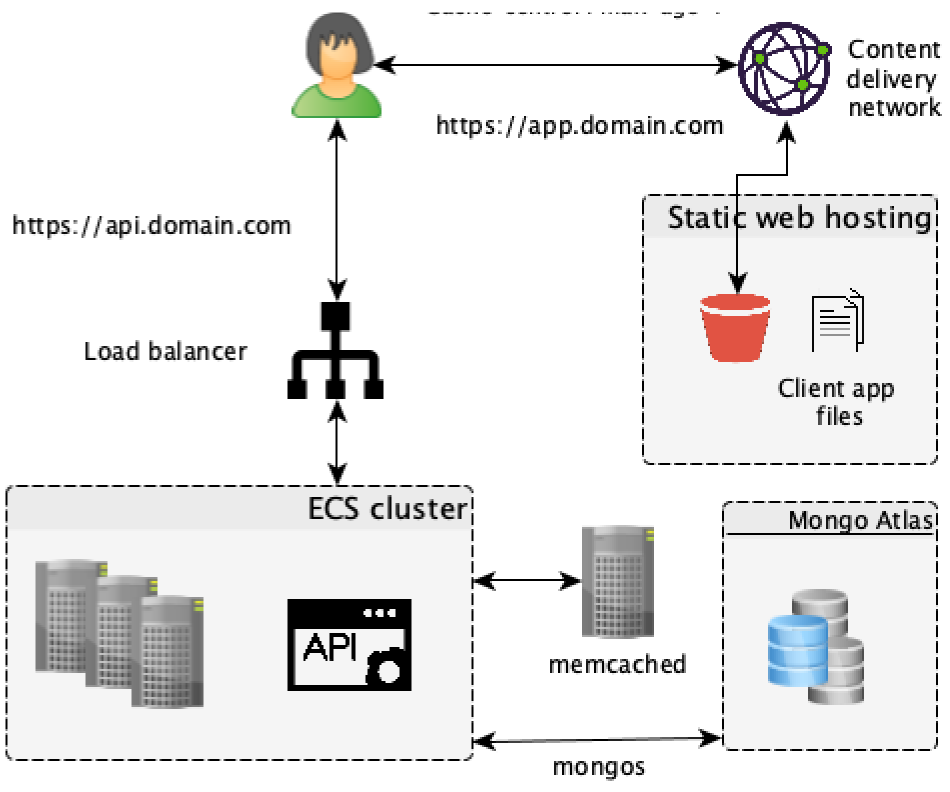
Everybody hates cache management, caching can happen at many of different layers, and cache-related issues are hard to reproduce, and a nightmare to debug.

Unfortunately the **performance** of distributed systems heavily relies on a **good caching strategy**. There are many good articles on good caching strategies so I won't go into much detail. Just know that if your **Static Web resources** are heavy, you'll probably want to take advantage of your user's browser cache by cleverly using the cache-control header.

If your user's facing pages are generated on the application servers over and over again, use a caching proxy like **Squid**. But most importantly, there is a high chance that you'll be making the same requests to your database over and over again. To lower your database load and save on the data transfer time, use a **memory object caching system** like **memcached** for objects that frequently utilized and rarely updated.

We started to consider using **memcached** because we frequently requested the same candidate profiles and job offers over and over again. Implementing it on a memory optimized machine **increased** our API **performance** by more than **30%** when we average all the requests response times in a day. Memcached is distributed as well, so it can run on different servers but still act like it's just one big memory space to store your objects.

Donate



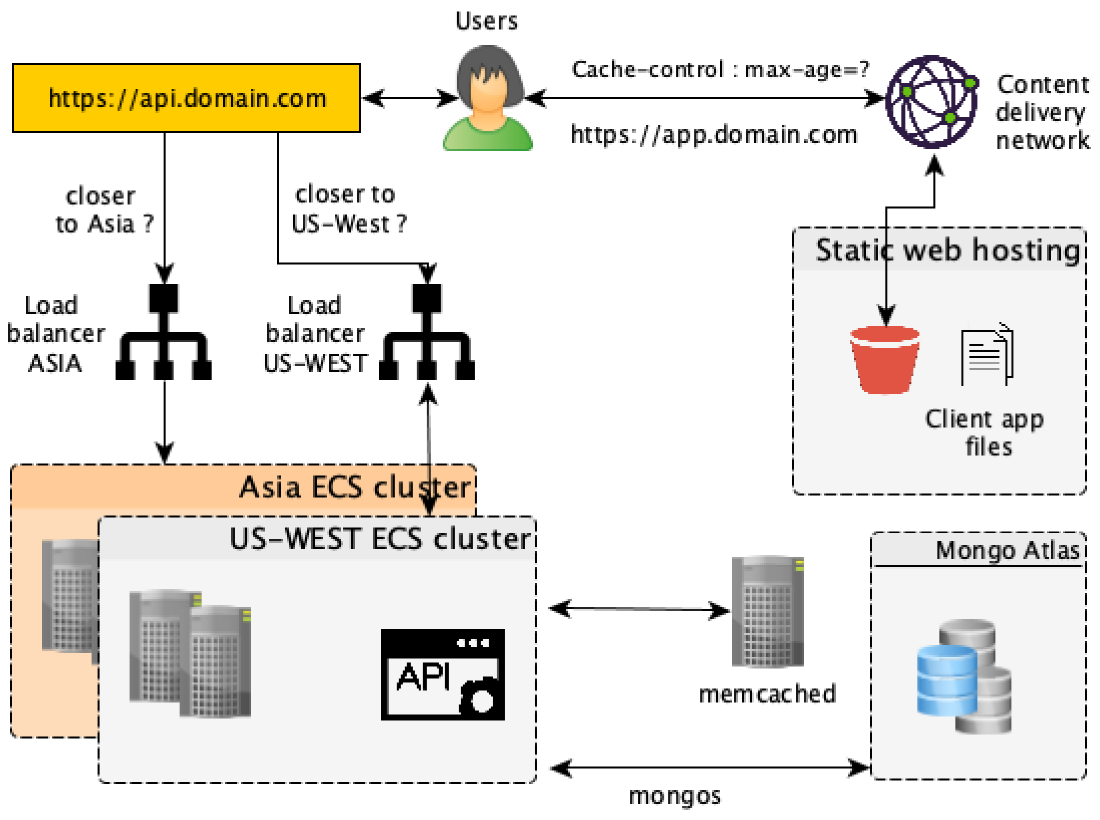
cache, cache everywhere

Location, location, location

Now we have a distributed system that doesn't have a **single point of failure** (if you consider AWS **ELBs** and a **distributed** memcached), and can **auto-scale** up and down. We also use **caching** to minimize network data transfers. Looks pretty good. At that point you probably want to **audit your third parties** to see if they will absorb the load as well as you.

slower for them, especially when they uploaded files. Indeed, even if our static web files were cached all over the world (courtesy of the CDN), all our application servers were deployed in the west of the US only. Users from **East Asia** experienced much **more latency** especially for big data transfers.

The solution was easy: **deploy** the exact same ECS cluster on a **new region** in Asia together with a new load balancer, and rely on Route 53 **Geoproximity Routing** to route users to the "nearest" load balancer. MongoDB **Atlas** also allows you to deploy your replicas **across regions** so there was no additional work required.



And here we are ! Our distributed system is ready.

Donate

Conclusion

While the distributed system you see here has been **simplified** for this post, we examined the parts you are most likely to see in a lot of modern web applications. Other topics related to but not covered are microservices architecture, file storage and encryption, database sharding, scheduled tasks, asynchronous parallel computing…maybe in the next post!

My main point is: **don't try to build the perfect system when you start** your product. Most of your design choices will be driven by what your product does and who is using it. You will only know that when you reach product market fit and start to have a good overview of your user base, and that can take months, years even.

Focus on **figuring out what people need**, and try to come up with a solution to their problem, even if it has a lot of **manual steps**. Then think about ways to **automate**, spend your time **coding** and **destroying**, and use **third parties** where it makes sense.

Don't scale but always think, code, and plan for scaling. Build your system **step by step**, don't address system design issues based on features that are not mature yet, and finally always try to find the best **trade-off** between the time you will spend and the gain in performance, money, and lowered risk.

If you liked this article and found any of it useful, hit that clap button and follow me for more architecture and development articles! ?

If this article was helpful, tweet it.

Donate

helped more than 40,000 people get jobs as developers.

Get started

Our Nonprofit

About

Alumni Network

Open Source

Shop

Support

Sponsors

Academic Honesty

Code of Conduct

Privacy Policy

Terms of Service

Copyright Policy

Trending Guides

2019 Web Developer Roadmap

Python Tutorial

CSS Flexbox Guide

JavaScript Tutorial

Python Example

HTML Tutorial

Linux Command Line Guide

JavaScript Example

Git Tutorial

React Tutorial

Java Tutorial

Linux Tutorial