alexpareto.com /scalability/systems/2020/02/03/scaling-100k.html

Scaling to 100k Users

10-13 minutes

Many startups have been there - what feels like legions of new users are signing up for accounts every day and the engineering team is scrambling to keep things running.

It's a good a problem to have, but information on how to take a web app from 0 to hundreds of thousands of users can be scarce. Usually solutions come from either massive fires popping up or by identifying bottlenecks (and often times both).

With that said, I've noticed that many of the main patterns for taking a side project to something highly scalable are relatively formulaic.

This is an attempt to distill the basics around that formula into writing. We're going to take our new photo sharing website, Graminsta, from 1 to 100k users.

1 User: 1 Machine

Nearly every application, be it a website or a mobile app - has three key components: an API, a database, and a client (usually an app or a website). The database stores persistent data. The API serves requests for and around that data. The client renders that data to the user.

I've found in modern application development that thinking of the client as a completely separate entity from the API makes it much easier to reason about scaling the application.

When we first start building the application, it's alright for all three of these things to run on one server. In a way, this resembles our development environment: one engineer runs the database, API, and client all on the same computer.

In theory, we could deploy this to the cloud on a single DigitalOcean Droplet or AWS EC2 instance like below:

With that said, if we expect Graminsta to be used by more than 1 person, it almost always makes sense to split out the database layer.

10 Users: Split out the Database Layer

Splitting out the database into a managed service like Amazon's RDS or Digital Ocean's Managed Database will serve us well for a long time. It's slightly more expensive than self-hosting on a single machine or EC2 instance - but with these services you get a lot of easy add ons out of the box that will come in handy down the line: multi-region redundancy, read replicas, automated backups, and more.

Here's what the Graminsta system looks like now:

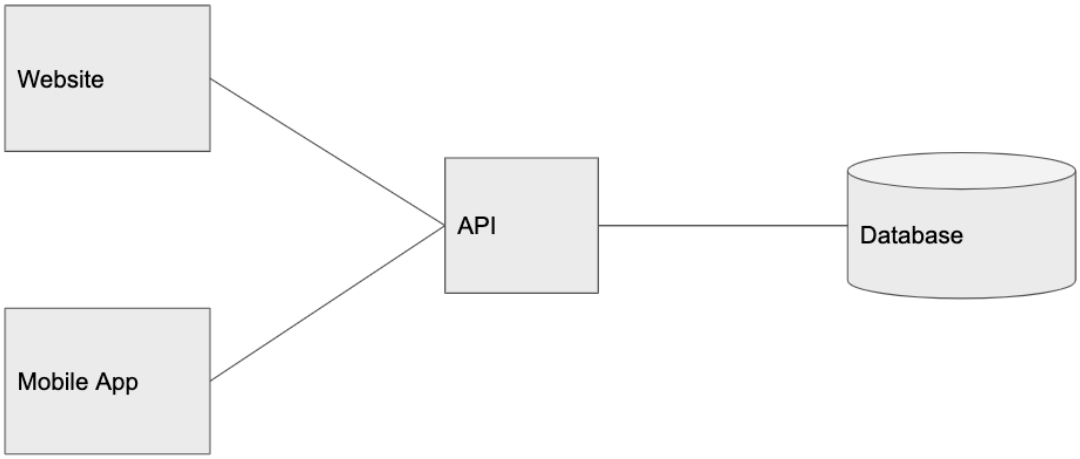
100 Users: Split Out the Clients

Lucky for us, our first few users love Graminsta. Now that traffic is starting to get more steady, it's time to split out the client. One thing to note is that **splitting out** entities is a key aspect of building a scalable application. As one part of the system gets more traffic, we can split it out so that we can handle scaling the service based on it's own specific traffic patterns.

This is why I like to think of the client as separate from the API. It makes it very easy to reason about building for multiple platforms: web, mobile web, iOS, Android, desktop apps, third party services etc. They're all just clients consuming the same API.

In the same vein, the biggest feedback we're getting from users is that they want Graminsta on their phones. So we're going to launch a mobile app while we're at it.

Here's what the system looks like now:



1,000 Users: Add a Load Balancer.

Things are picking up at Graminsta. Users are uploading photos left and right. We're starting to get more sign ups. Our lonely API instance is having trouble keeping up with all the traffic. We need more compute power!

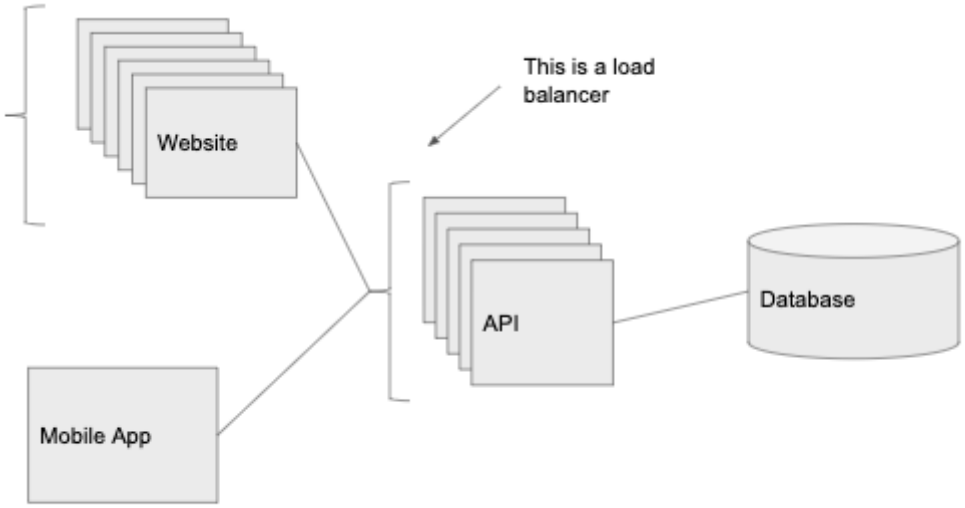
Load balancers are very powerful. The key idea is that we place a load balancer in front of the API and it will route traffic to an instance of that service. This allows for horizontal scaling (increasing the amount of requests we can handle by adding more servers running the same code).

We're going to place a separate load balancer in front of our web client and our API. This means we can have multiple instances running our API and web client code. The load balancer will route requests to whichever instance has the least traffic.

What we also get out of this is redundancy. When one instance goes down (maybe it gets overloaded or crashes), then we have other instances still up to respond to incoming requests - instead of the whole system going down.

A load balancer also enables autoscaling. We can set up our load balancer to increase the number of instances during the Superbowl when everyone is online and decrease the number of instances when all of our users are asleep.

With a load balancer, our API layer can scale to practically infinity, we will just keep adding instances as we get more requests.



Side note: At this point what we have so far is very similar to what PaaS companies like Heroku or AWS's Elastic Beanstalk provide out of the box (and why they're so popular). Heroku puts the database on a separate host, manages a load balancer with autoscaling, and lets you host your web client separately from your API. This is a great reason to use a service like Heroku for projects or early stage startups - all of the necessary basics come out of the box.

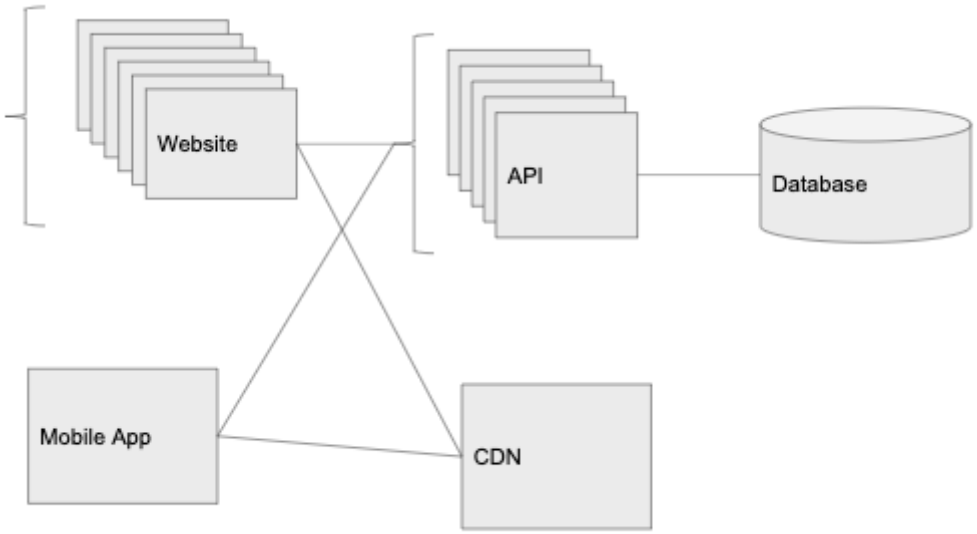
10,000 Users: CDN

We probably should have done this from the beginning, but we're moving fast here at Graminsta. Serving and uploading all these images is starting to put way too much load on our servers.

We should be using a cloud storage service to host static content at this point: think images, videos, and more (AWS's S3 or Digital Ocean's Spaces). In general, our API should avoid handling things like serving images and image uploads.

The other thing we get out of a cloud storage service is a CDN (in AWS this is an add-on called Cloudfront, but many cloud storage services offer it out of the box). A CDN will automatically cache our images at different data centers throughout the world.

While our main data center may be hosted in Ohio, if someone requests an image from Japan, our cloud provider will make a copy and store it at their data center in Japan. The next person to request that image in Japan will then recieve it much faster. This is important when we need to serve larger file sizes like images or videos that take a long time to load + send across the world.



100,000 Users: Scaling the Data Layer

The CDN helped us out a lot - things are booming at Graminsta. A YouTube celebrity, Mavid Mobrick, just signed up and posted us on their story. The API CPU and memory usage is low across the board - thanks to our load balancer adding 10 API instances to the environment - but we're starting to get a lot of timeouts on requests…why is everything taking so long?

After some digging we see it: the Database CPU is hovering at 80-90%. We're maxed out.

Scaling the data layer is probably the trickiest part of the equation. While for API servers serving stateless requests, we can merely add more instances, the same is not true with *most* database systems. In this case, we're going to explore the popular relational database systems (PostgreSQL, MySQL, etc.).

Caching

One of the easiest ways to get more out of our database is by introducing a new component to the system: the cache layer. The most common way to implement a cache is by using an in-memory key value store like Redis or Memcached. Most clouds have a managed version of these services: Elasticache on AWS and Memorystore on Google Cloud.

The cache comes in handy when the service is making lots of repeated calls to the database for the same information. Essentially we hit the database once, save the information in the cache, and never have to touch the database again.

For example, in Graminsta every time someone goes to Mavid Mobrick's profile page, the API layer requests Mavid Mobrick's profile information from the database. This is happening over and over again. Since Mavid Mobrick's profile information isn't changing on every request, that info is a great candidate to cache.

We'll cache the result from the database in Redis under the key `user:id` with an expiration time of 30 seconds. Now when someone goes to Mavid Mobrick's profile, we check Redis first and just serve the data straight out of Redis if it exists. Despite Mavid Mobrick being the most popular on the site, requesting the profile puts hardly any load on our database.

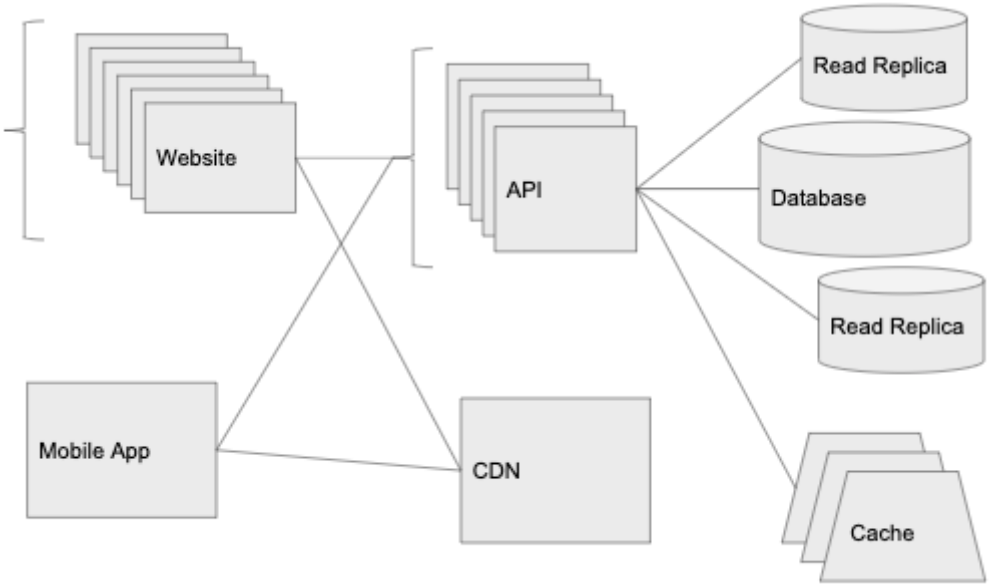
The other plus of most cache services, is that we can scale them out easier than a database. Redis has a built in Redis Cluster mode that, in a similar way to a load balancer[1], lets us distribute our Redis cache across multiple machines (thousands if one so pleases).

Nearly all highly scaled applications take ample advantage of caching, it's an absolutely integral part of making a fast API. Better queries and more performant code are all a part of the equation, but without a cache none of these will be sufficient to scale to millions of users.

Read Replicas

The other thing we can do now that our database has started to get hit quite a bit, is to add read replicas using our database management system. With the managed services above, this can be done in one-click. A read replica will stay up to date with your master DB, and will be able to be used for SELECT statements.

Here's our system now:



Beyond

As the app continues to scale, we're going to want to focus on splitting out services that can be scaled independently. For example, if we start to make use of websockets, it would make sense to pull out our websocket handling code. We can put this on new instaces behind their own load balancer that can scale up and down based on how many websocket connections have been opened or closed, independently of how many HTTP requests we have coming in.

We're also going to continue to bump up against limitations on the data layer. This is when we are going to want to start looking into partitioning and sharding the database. These both require more overhead, but effectively allow the data layer to scale infinitely.

We will want to make sure that we have monitoring installed using a service like New Relic or Datadog. This will ensure we understand what requests are slow and where improvement needs to be made. As we scale we will want to be focused on finding bottlenecks and fixing them - often by taking advantage of some of the ideas in the previous sections.

Hopefully at this point, we have some other people on the team to help out as well!

References:

This post was inspired by one of my favorite posts on High Scalability. I wanted to flesh the article out a bit more for the early stages and make it a bit more cloud agnostic. Definitely check it out if you're interested in these kind of things.

Footnotes