How scaling works

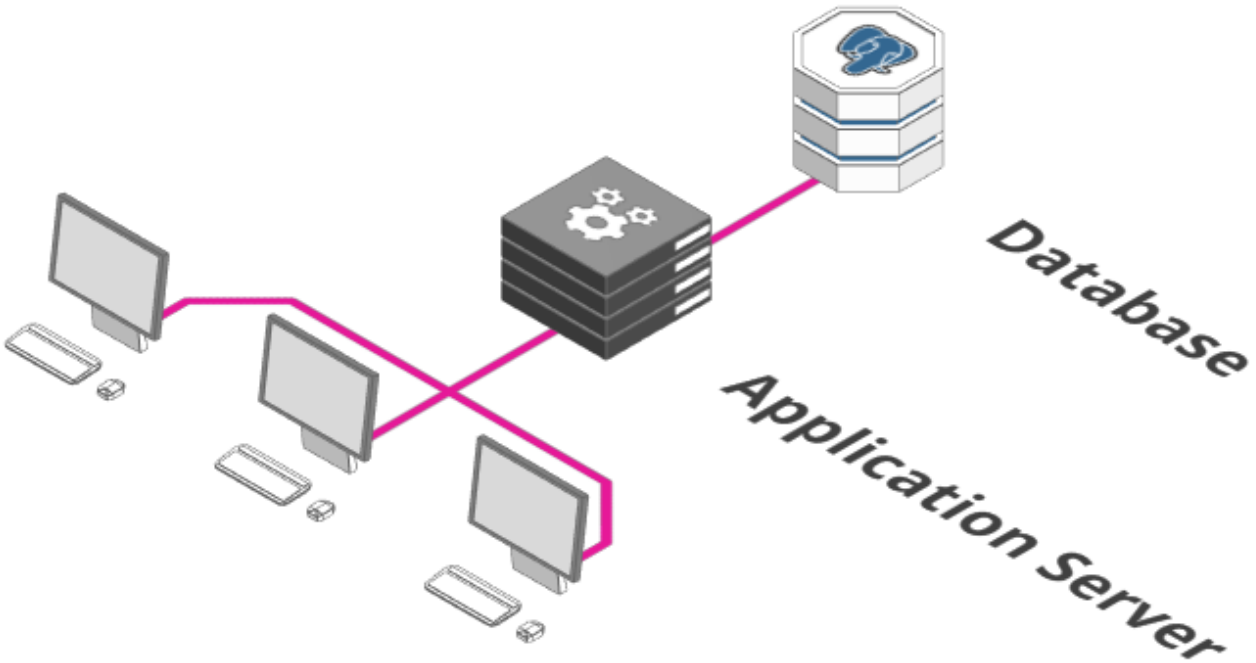
A few years ago, this post would have started with a discussion of "vertical" vs "horizontal" scaling (also called scaling up vs scaling out). In a nutshell, vertical scaling means running the same thing on a more powerful computer whereas horizontal scaling means running many processes in parallel.

Today, almost no one is scaling up / vertically anymore. The reasons are simple:

- computers get exponentially more expensive the more powerful they are
- a single computer can only be so fast, putting a hard limit on how far one can scale vertically
- multi-core CPUs mean that even a single computer is effectively parallel- so why not parallelize from the start?

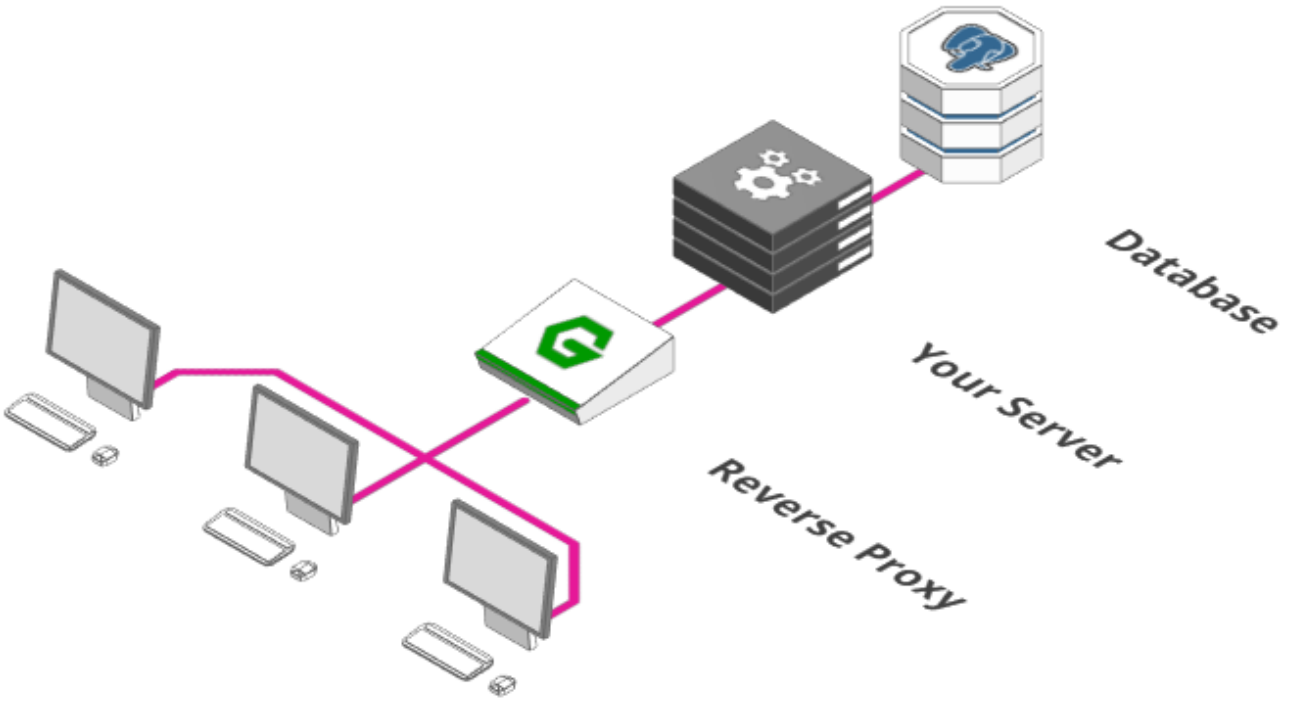
Alright, horizontal scaling it is! But what are the required steps?

1. A single server + database



This is probably how your backend looks initially. A single application server running your business logic and a database that stores data for the long run. Things are nice and simple, but the only way for this setup to cater for higher demand is to run it on a beefier computer - not good.

2. Adding a Reverse Proxy

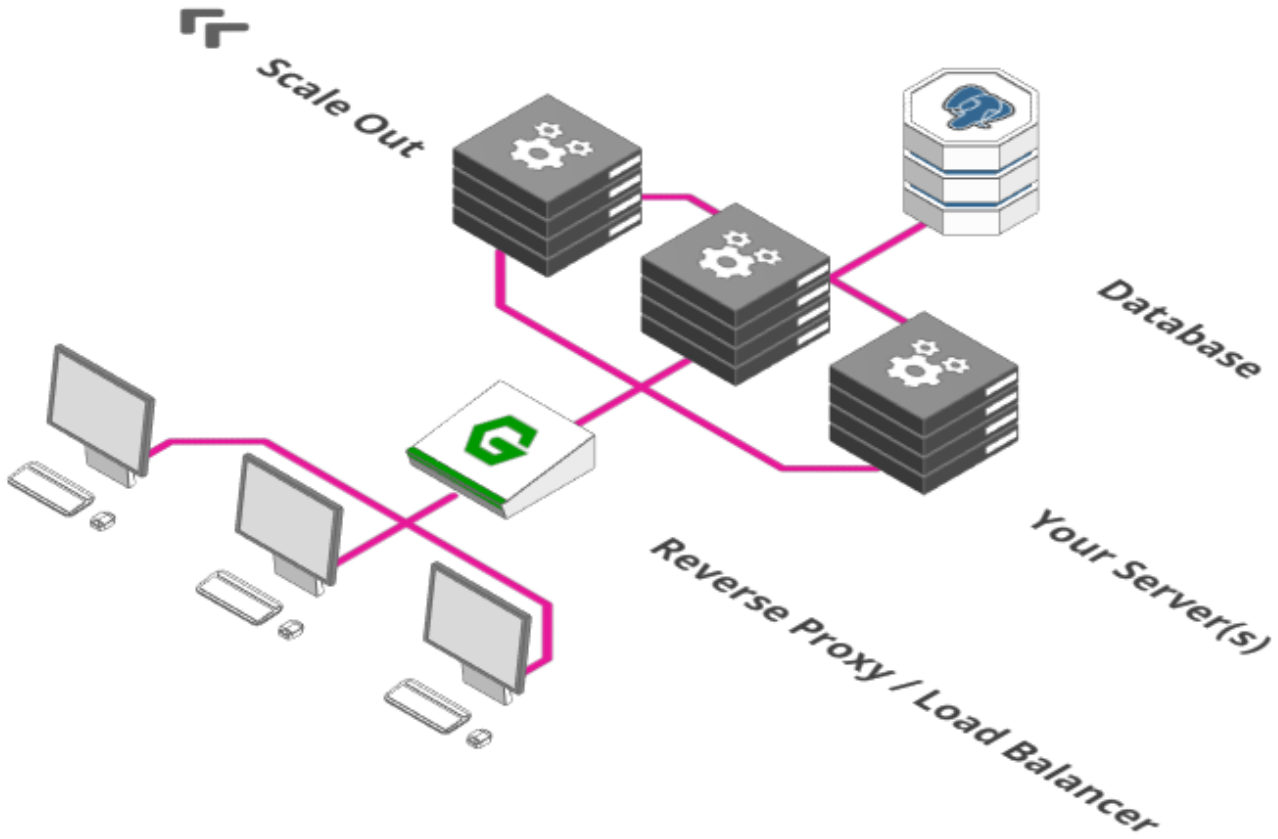


An initial step to prepare your architecture for larger scale is to add a "reverse proxy". Think of it as the reception desk in a hotel. Sure, you could just let guests go directly to their rooms - but really, what you want is an intermediary that checks if a guest is allowed to enter, has all her papers in order and is on the way to a room that actually exists. And should a room be closed, you want someone telling the guest in a friendly voice, rather than just having them run into limbo. That's exactly what a reverse proxy does. A proxy, in general, is just a process that receives and forwards requests. Usually, these requests would go from our server out to the internet. This time, however, the request comes from the internet and needs to be routed to our server, so we call it a "reverse proxy".

Such a proxy fulfills a number of tasks:

- Health Checks make sure that our actual server is still up and running
- Routing forwards a request to the right endpoint
- Authentication makes sure that a user is actually permissioned to access the server
- Firewalling ensure that users only have access to the parts of our network they are allowed to use ... and more

3. Introducing a Load Balancer

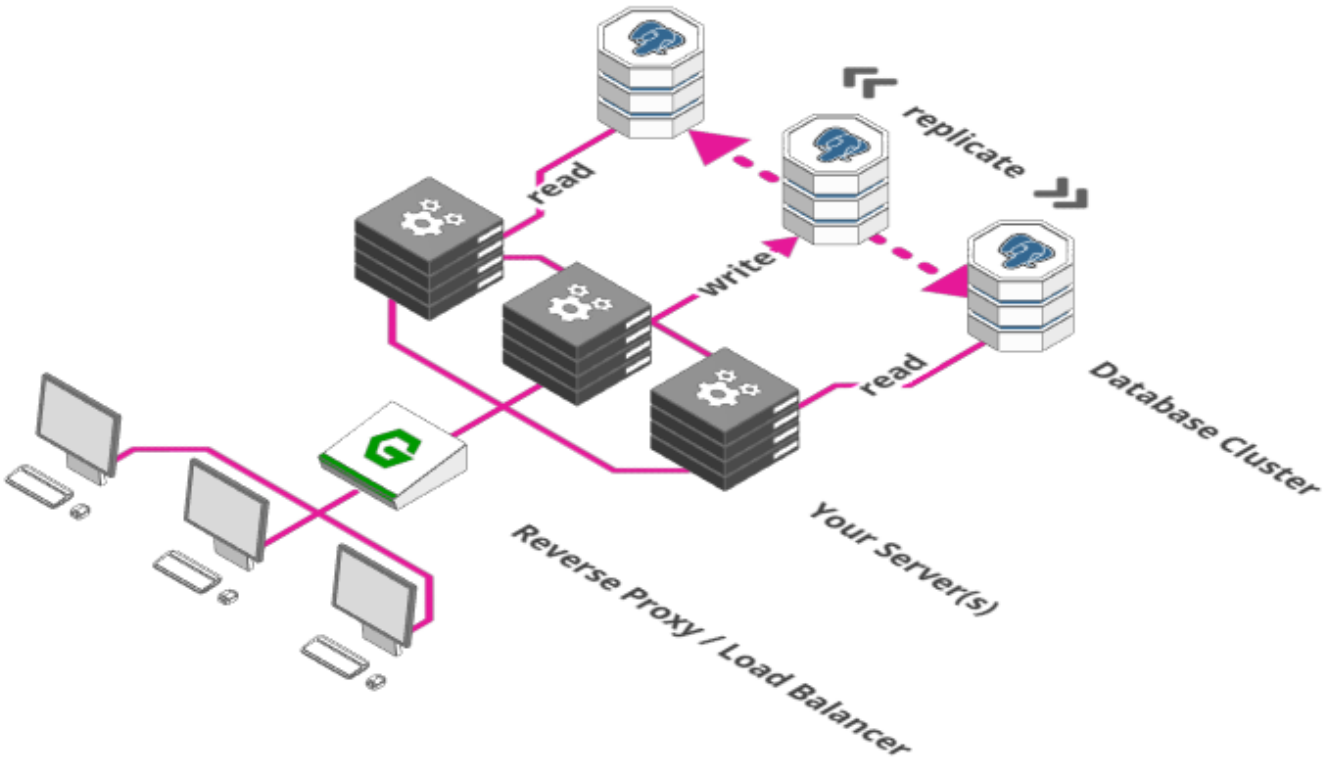


Most "Reverse Proxies" have one more trick up their sleeve: they can also act as load balancers. A load balancer is a simple concept: Imagine a hundred users are ready to pay in your online shop in a given minute. Unfortunately, your payment server can only process 50 payments in the same time. The solution? You simply run two payment servers at once.

A load balancer's job is now to split incoming payment requests between these two servers. User one goes left, user two goes right, user three goes left and so on.

And what do you do if five-hundred users want to pay at once? Exactly, you scale to ten payment servers and leave it to the load balancer to distribute the incoming requests between them.

4. Growing your Database



Using a load balancer allows us to split the load between many servers. But can you spot the problem? While we can utilize tens, hundreds or even thousands of servers to process our requests, they all store and retrieve data from the same database.

So can't we just scale the database the same way? Unfortunately not. The problem here is consistency. All parts of our system need to agree on the data they are using. Inconsistent data leads to all sorts of hairy problems - orders being executed multiple times, two payments of $90 being deducted from an account holding $100 and so on... so how do we grow our database while ensuring consistency?

The first thing we can do is split it into multiple parts. One part is exclusively responsible for receiving data and storing it, all other parts are responsible for retrieving the stored data. This solution is sometimes called a Master/Slave setup or Write with Read-replicas. The assumption is that servers read from the database way more often than they write to it. The good thing about this solution is that consistency is guaranteed since data is only ever written to a single instance and flows from there in one direction, from write to read. The downside is that we still only have a single database instance to write to. That's ok for small to medium web projects, but if you run Facebook it won't do. We'll look into further steps to scale our DB in chapter 9.

5. Microservices