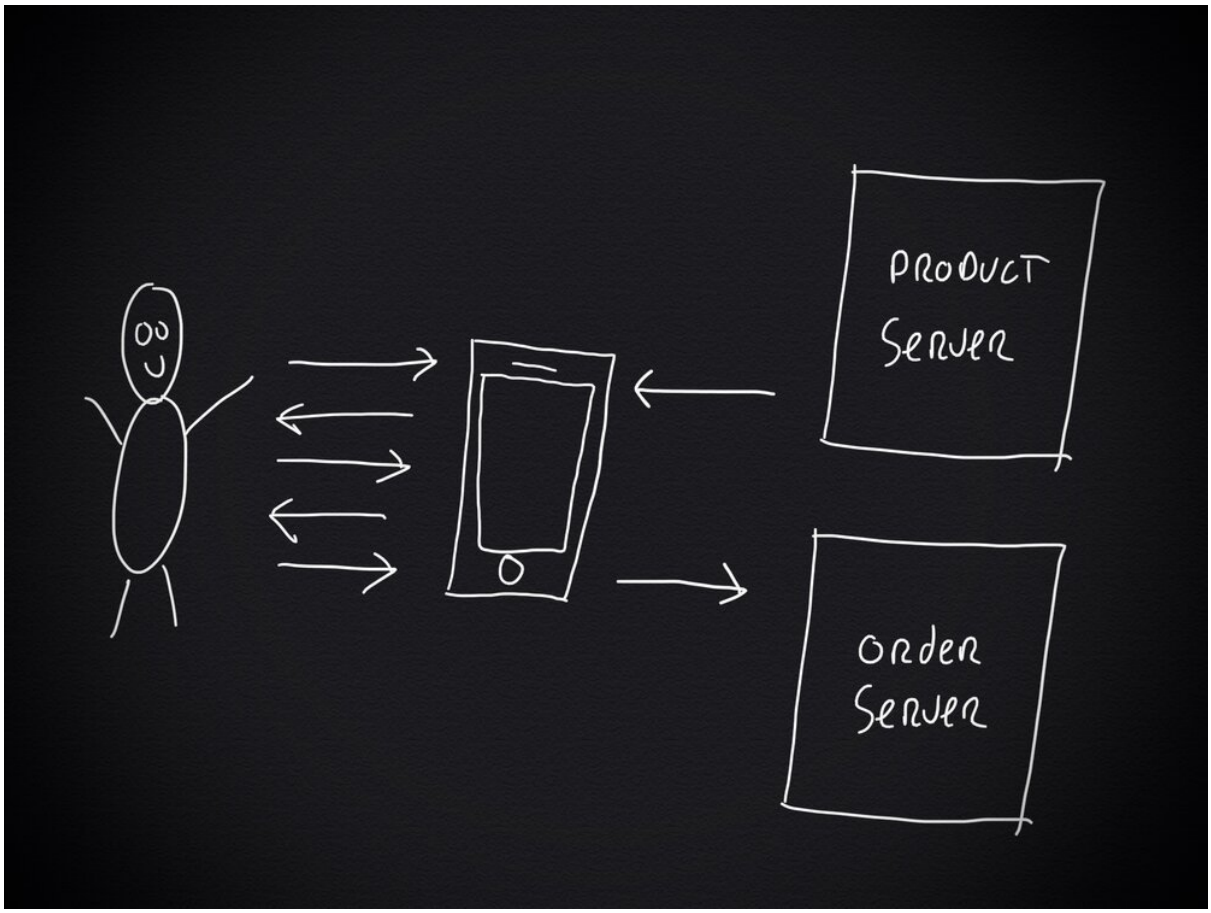
Scalable application design without magic

Leveraging client computing power for high performance with many users

Willem L. Middelkoop
May 11, 2020

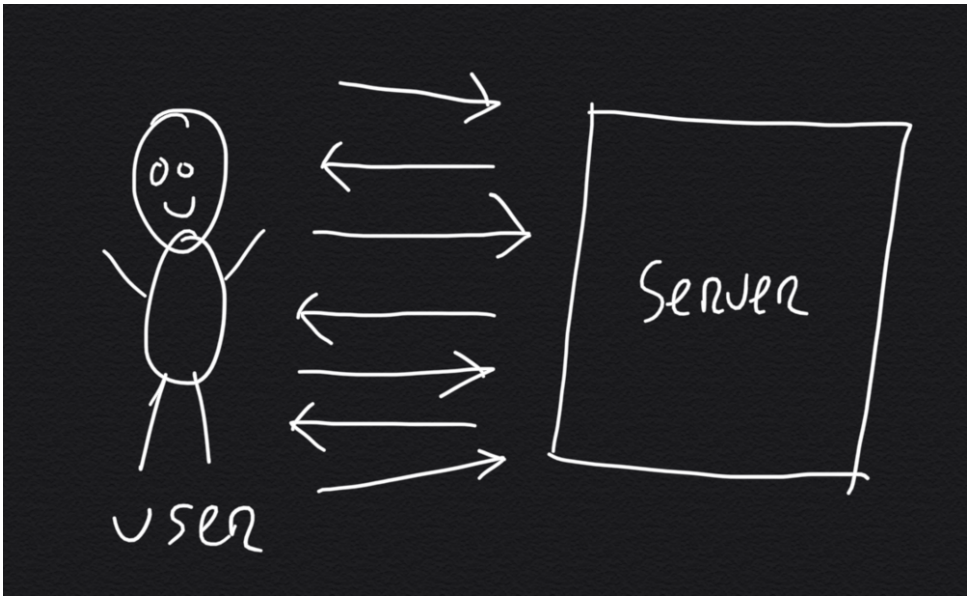


As part of the online food ordering app I'm building, I needed to design a scalable backend infrastructure that could handle lots of concurrent users. Scalability is considered a hard problem to tackle. Often it's presented like it's something magical, done by million dollar companies using secret tools. But, there is no such thing as magic, or is there?

What is scalability?

If you're building an app, you probably start very small. Just a few users (or only you, the developer) that work with the app. Everything will be jolly good: the app works

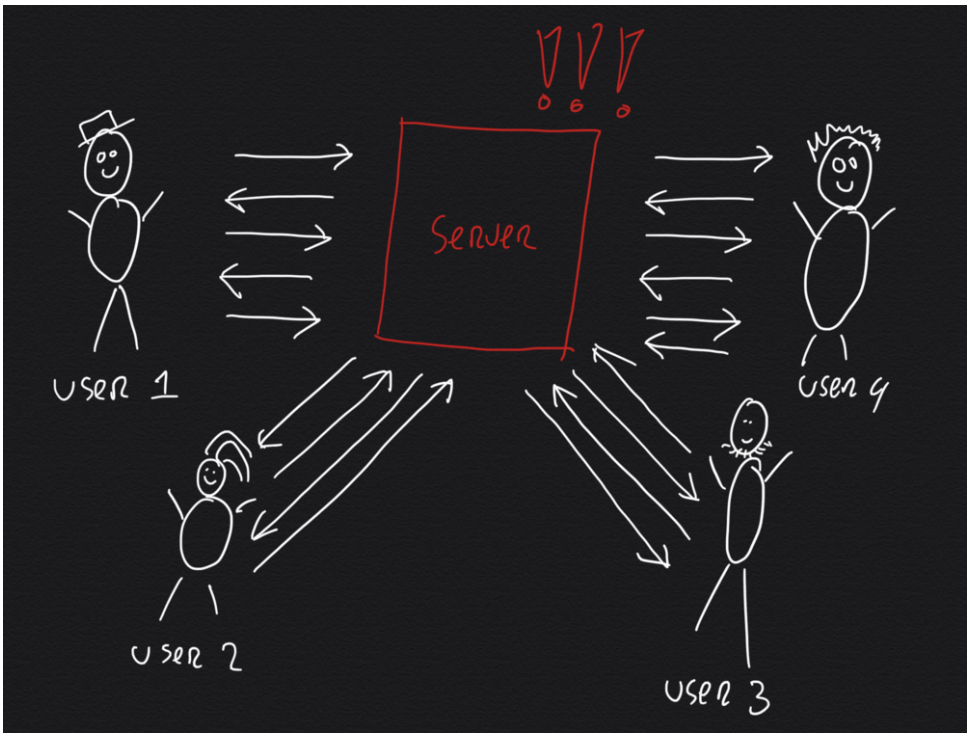
well, fast and there are no problems. *(cherish this feeling!)*



One user interacting with one server

In this early and small scenario the single server that runs the app can handle the user interacting with it. Every time the user taps or clicks a button, the single server does some work. If you program well, these workloads can be optimised into efficient chunks of work. Doing it like this is a typical example of a monolithic software architecture.

Even if your project or app is rather small, at some point there may be an influx of users. If your system can't support high loads, it has high chances of failure.



Server under high load handling multiple concurrent users

Because of the monolithic app design, all users interact with the same server. All their interaction (taps, clicks, input) is handled by this server. With an increased load it has to work a lot harder. At a certain point, you will notice things slowing down because the server cannot keep up with the amount of work it has to do. This means your app does not scale well, in other words: you're in trouble.

Failure on 'moment supreme'

Not scaling well is a big problem that nobody should underestimate. Most apps (and their business models) rely on high volumes with little revenue per individual user. If you want your app to be a (financial) success, you need to make sure it works well when there is a sudden increase of traffic. If your app crashes when it suddenly gets the attention of the masses, you'll miss a 'one-in-a-lifetime' opportunity for (organic) growth.

In a way designing for scalability is like keeping your seatbelts fastened when you're sitting in your self-built rocket ship, featuring experimental engines. You never quite know exactly when it will ignite, but when it does you'd better make sure you sit tight (and enjoy the ride!).

Common ways to achieve scalability

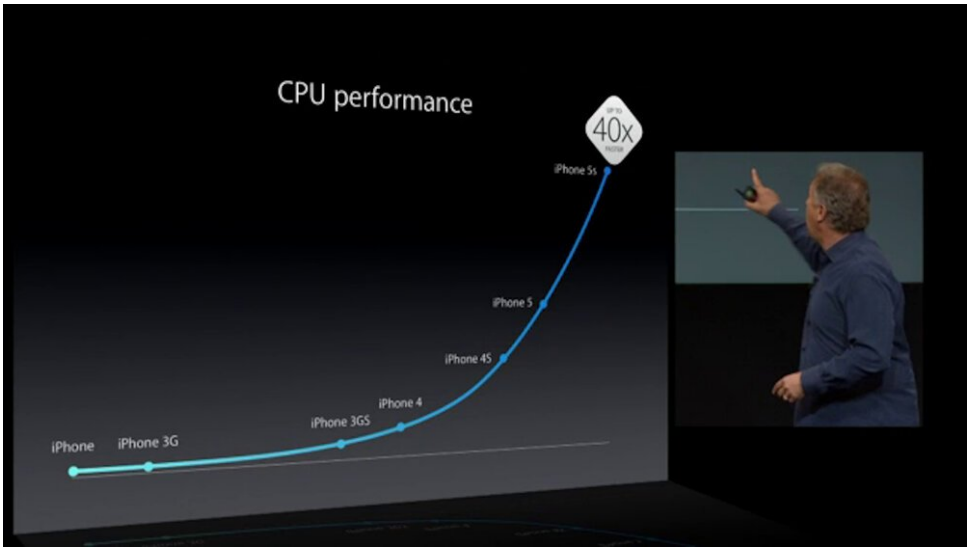
A lot has been written on building scalable apps, common ways to improve scalability are:

- **Increase server capacity**: add more memory, CPU power, storage. This will work, but it will only get you 'so far'.
- **Add more servers**: instead of a single server acting as a bottle neck, you can add more servers that will share the workload. This is easier said than done as it will require load balancing and data sharing which can be tricky (e.g. if you have an order counter, and two servers are counting, which one determines the next number?)
- **Optimise programming code**: use the right tools! Choose a performance oriented programming language and ditto server software. Look into functional programming to allow your code to run on multiple cores asynchronously. Make it stateless. Benchmark it. Optimise it. Every millisecond gained on a single request adds up quickly when you're handling big volumes.
- **Use the database and separate it from the application server**: If you're using a database, leverage its power! Take advantage of query caching, indexes and search capabilities. Most database solutions offer battle-proof clustering options; don't try to invent the wheel that others have perfected already.

The best thing you can do is to consider all of these options. Even if you're not adding more servers right away, you should write your code to support it later on. Benchmarking and optimising your code should be an integral part of your work, not just an afterthought.

Leveraging client computing power

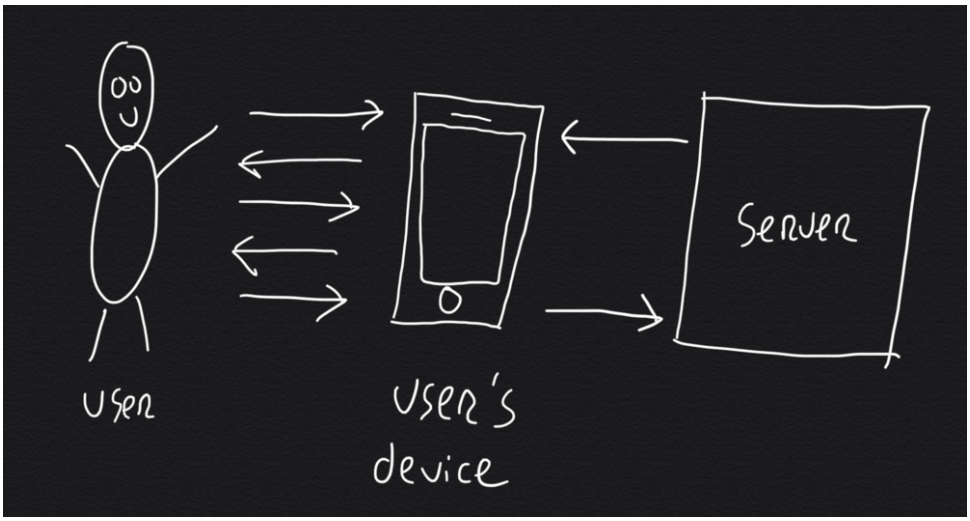
When you have considered all the common ways to achieve scalability, there is ”one more thing”.



Smartphones (and tablets, PC's) have become more powerful over the years (image: Apple)

Few developers consider this: the scalability problem contains the solution! When you have 1000 users, you have 1000 computers with powerful CPU's and lots of memory. Modern computers and smartphones have become increasingly capable.

You simply have to find a way to tap into the client's computing power. While it may sound like an evil thing (using the user's computing power), it will provide them with a much better, faster, experience. What's best is that *their* computing power will be spent on *their* fantastic experience, solving your scalability problem as happy side effect.



Leveraging a user's device: the user's phone does most of the work originally done by the server

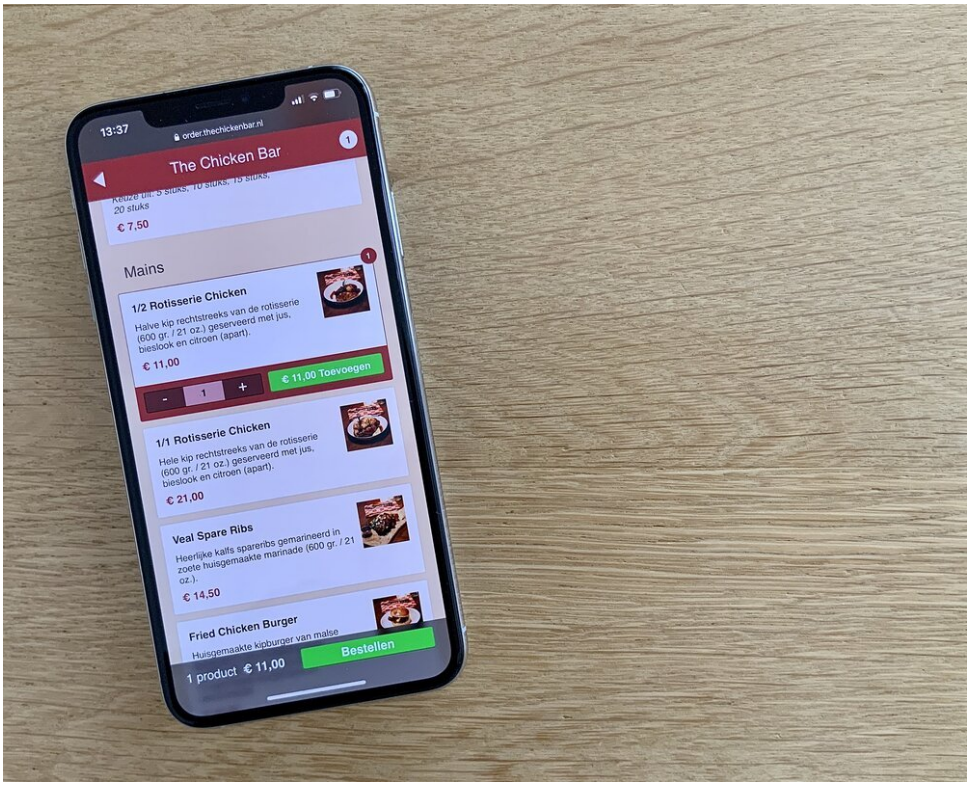
By leveraging the user's computing power, you reduce the amount of work the server has to do. Instead of the server doing a little work *every* time the user taps or clicks a button, this work is now mostly done by the user's device. This is more efficient than communicating over a Wifi/4G connection with a server (saving battery power and time).

It requires a little re-thinking, but this principle can be applied to web applications too. Instead of handling all things on the server, you can perform a lot of work on the

client. Things like searching, browsing, filtering, navigating and even adding things to a shop cart can be done using client side JavaScript. You - dear developer - have to look for ways to make this work, but if you succeed you'll have near infinite scalability at your fingertips.

In practice: food ordering app

For the food ordering app I looked for ways to leverage the client's computing power to achieve high performance. The main challenge with delivery and takeaway orders is that they peak around diner time. Lots of people use the app around the same time, this is a recipe for scalability problems.

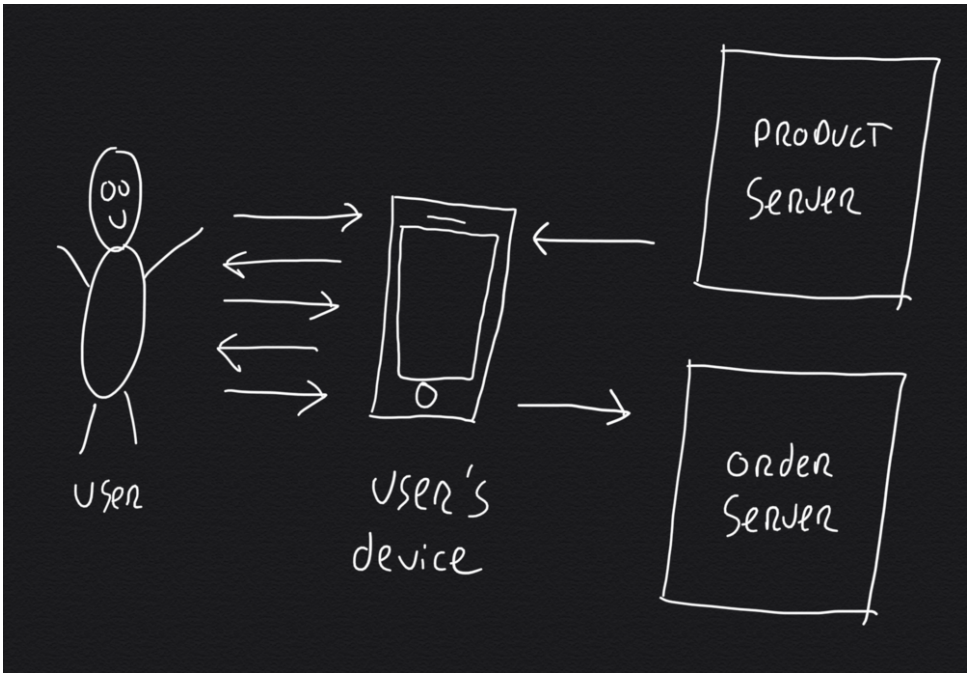


Ordering food online

If you think of ordering food online, you can break the entire process into smaller chunks:

- **1) Opening the page**: loading the app/page
- **2) Browsing the different options**: listing the products, seeing their descriptions, prices, photo's, etc.
- **3) Searching for something specific**: search by category, product name, etc.
- **4) Selecting a product**: adding it to your order
- **5) Customising a product**: selecting a sauce, side-dish, topping, etc.
- **6) Entering your information**: your name, phone number, delivery details, etc.
- **7) Paying for your order**: selecting a payment method, connecting to your bank or credit card
- **8) Closing the app**

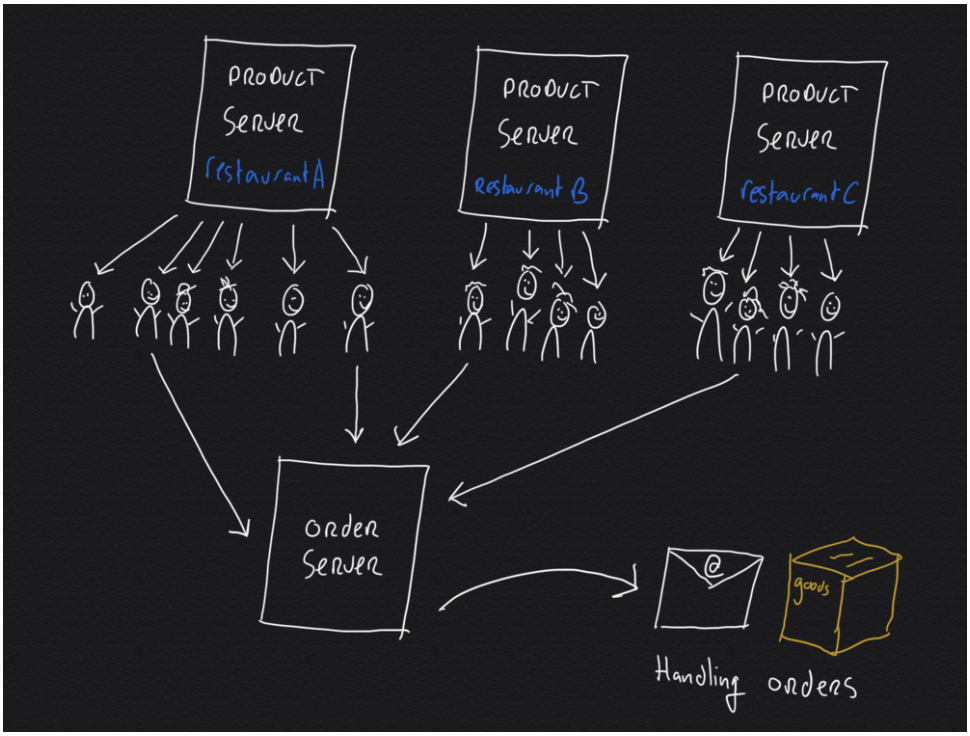
Most of these steps can be programmed in such a way that the server is *not necessary*. Only step 1 (**loading**) and step 7 (**paying**) require contact with the backend infrastructure. It is important to realise that steps 2 to 5 are repeated multiple times, as it's common for users to add multiple products to their order.



Separating the remaining server workload: serving products and handling orders

By separating the remaining workload by task, the software can be further optimised. The **"product server"** is optimised to serve static assets (texts, prices, images) that allow the client to create the product browsing experience. You can highly optimise this kind of server, by leveraging caching, HTTP/2, compression, etc.

Only those users that complete the ordering process will be forwarded to the **"order server"** which handles the payment. It will connect to the payment provider. Once a payment is completed, the payment provider will contact the order server to inform it about the payment status. This is something you want to take full (server side) control over, as I did when I designed a payment system earlier. The remaining workload on the "order server" is much smaller as not everybody will place an order, and much of its order processing work can be done in the background.



Handling lots of traffic through distributed computing

To maximise scalability you can easily add multiple servers (or use a content delivery network) to serve the static assets. For the food ordering app I use a separate "product server" *per restaurant*. This enables high performance and reduces loading times. People notice it's fast, some can't believe it's web technology, it's *almost* magical.

Conclusion

Designing a scalable application can be done without magic or million dollar budgets. You just have to think it through and look for opportunities that present themselves along with the challenges.

The problem contains the solution. Dealing with lots of concurrent users is not just a problem: they bring their own computing power, it's the solution! You just have to use it.