A cookbook inspired by real-world solutions and *scribbled* sticky notes

NGINX *by* Example

VOLUME 1

JASON JOSEPH NATHAN

Optimized by Humans. Approved by Servers.

NGINX by Example Vol. 1

First edition.

Published in Singapore.

Published under the Geekist.co imprint.
Set in Lora & Ysabeau Office and designed using Adobe InDesign.

Dedicated to my children

the innocent

caught in the chaos of others

Preface

I've been using NGINX for years now, sometime in the mid to late 2000s, back when Igor Sysoev was still actively maintaining it, just before F5 took over.

Even then, the value was obvious: speed. I remember testing it - Apache on a fresh LAMP stack versus NGINX with PHP - and seeing page load times drop from 200ms to 20ms out of the box. That alone sold me, and the rest is history.

Decades later, there are plenty of books on NGINX and I've read more than a few. But I always found them either too academic or too full of jargon for what is already a pretty dry topic. I wanted something that explained *how* things work, but also *why* and *when*, without losing someone in the weeds.

That's easier said than done. Writing this book meant accounting for real-world scenarios and edge cases, while keeping things readable. So this is my attempt at organizing years of scribbled notes into something practical and (hopefully) enjoyable.

Volume I is split into three chapters, each covering a specific theme. Within each topic, I've framed a problem, introduced a usable pattern, and then expanded on its variations and caveats.

Yes, this is a collection of battle-tested notes, but also a refinement of everything I've learned from other books & docs and late-night experiments, updated to reflect today's best practices.

You'll find companion code and configurations at:

https://geekist.co/nginx-by-example

To follow updates from the broader Geekist ecosystem:

https://geekist.co/subscribe/

I hope you enjoy reading this as much as I've enjoyed writing it.

Contents

CHAPTER I

The Hidden Power in Your Web Server

Introduction

For many of us, HTTPS feels like a solved problem. You run Certbot, the browser gives you a reassuring padlock and you move on. Sometimes your hosting provider your CDN abstracts it away and in most cases, that's actually good enough.

Until it *isn't*.

One day, something breaks. A customer signs up and their subdomain fails to load, or a QA environment stops working because the server refuses the connection. Sometimes (especially with Certbot), a certificate expires and nobody notices until dashboards across the board stop rendering.

You dig through the configs and discover a mess of inconsistencies: duplicated blocks, mismatched headers, SSL paths pointing to the wrong files, even a missing redirect or two. What was once a clean setup becomes fragile patchwork built on best intentions and rushed fixes.

This is where most developers get stuck. They treat NGINX as a simple router or static file server or a reverse proxy with just enough brains to get requests where they need to go. But NGINX is far more capable than that. It can manage trust. It can negotiate identity. It can serve as a rules engine, a security layer and a control tower, all in one place.

This chapter is about rediscovering that power.

We're not just going to turn on HTTPS, we're going to design for it. Our configs will be structured so that the secure path is the path of least resistance.

TLS parameters will be centralized, redirects made consistent and every subdomain - from staging to production - will inherit the same best practices by default.

We'll go deeper, too. We'll explore mutual TLS for scenarios where trust isn't one-way and build tenant-aware routing with wildcard subdomains. And we'll do it all in a way that's maintainable and scalable - and surprisingly clean.

It's a secure, resilient foundation, using the tools you already have but probably aren't using fully.

The goal isn't to impress with clever tricks. It's to sleep better at night, knowing your infrastructure has your back.

Let's begin.

How OCSP Stapling Makes Browsers Trust You Faster

When you secure your site with HTTPS, you're not just encrypting traffic. You're making a promise. That certificate? It tells browsers and users that this domain belongs to you and that a trusted authority verified it.

But trust is *fragile*.

If your certificate gets revoked (compromise maybe or expiration), you want browsers to know - and fast. And you want that check to happen without adding delays to every page load.

That's where OCSP stapling comes in. It's your way of saying: "Here's proof that this certificate is still valid. Don't just trust me... here's the latest response from the issuer."

It matters even more with wildcard certificates because if trust breaks once, it breaks for every subdomain you own.

The Problem

Every modern browser supports the **Online Certificate Status Protocol** (OCSP) but by default they must contact the certificate authority to check whether your cert has been revoked. This introduces a few issues:

- A live internet connection is needed the first time your site loads
- The CA's OCSP server becomes a single point of failure
- IIf it's slow or unreachable, your site appears broken or un-trusted

Worse, some browsers "soft fail," meaning they allow the connection even if the check fails, defeating the point of revocation.

The Pattern

You only need to configure stapling once. By placing these directives in your global SSL include file, every subdomain benefits automatically.

The Implementation

In this chapter, we'd often refer to a shared `ssl_params` file, which is a central place to define TLS settings and certificate logic for all your server blocks. Here's how that file might look with OCSP stapling configured:

```
# /etc/nginx/ssl_params
ssl_certificate /etc/ssl/certs/example_com.crt;
ssl_certificate_key /etc/ssl/private/example_com.key;

ssl_protocols TLSv1.2 TLSv1.3;
ssl_prefer_server_ciphers on;
ssl_ciphers EECDH+AESGCM:EDH+AESGCM;

ssl_stapling on;
ssl_stapling_verify on;

ssl_trusted_certificate /etc/ssl/certs/ca-bundle.crt;
resolver 8.8.8.8 1.1.1.1 valid=300s;
resolver_timeout 5s;
```

Let's break this down.

- \`ssl_stapling on;\` enables stapling support.
- \`ssl_stapling_verify on;\` ensures the response is cryptographically valid.
- \`ssl_trusted_certificate\` must point to a CA bundle that includes the issuer of your cert or stapling eill fail silent

Exceptions and Variants

You might encounter setups where OCSP stapling does not work as expected. Some common issues are:

1. Your certificate authority does not support OCSP
2. The intermediate certs were not installed correctly
3. The CA's response URI is unreachable from your server

4. You're using Let's Encrypt with certs obtained via DNS challenge and missing intermediate chains

To debug stapling, use:

```
openssl s_client -connect example.com:443 -status
```

You should see an OCSP response block. If you don't, something's wrong.

Also note: OCSP responses are cached. If you restart NGINX too frequently, you may see temporary failures.

Testing the Setup

Run the following after you've configured and reloaded NGINX:

```
curl -v https://localhost:8443 --cacert ./ca.crt
```

Look for the output:

```
OCSP stapling test passed.
```

Followed by response data. If it's missing entirely, stapling isn't working.

Operational Notes

- OCSP responses are usually valid for 4 to 7 days. NGINX caches them in memory, so there's no need to re-fetch on every request.
- If you rotate your certificate, the OCSP cache is flushed. The next TLS request will trigger a fresh lookup.
- If your server cannot reach the CA's OCSP endpoint, stapling will fail silently unless you log it explicitly.
- Some CDNs automatically staple OCSP responses. If you're behind one, your origin does not need to do it but it's a good practice anyway.

Summary

OCSP stapling is one of those things most people forget to configure, even though it's almost always supported by modern certs, browsers & web servers.

When you're using a wildcard certificate, the stakes are even higher. That one certificate holds the keys to every subdomain you serve. If something goes wrong, it affects all of them.

By enabling OCSP stapling and doing it in a central, shared location, you're not just improving performance, you're showing every visitor that trust still holds.

You're proving it before they even have to ask.

One SSL Config to Rule All Your Subdomains

It always starts small. A subdomain here, another one there. Maybe it is a new blog.example.com or an app you are testing. A dev tool sneaks in, followed by a quick client preview. Before you even realize it, your once-pristine NGINX config becomes a patchwork of near-duplicates. Each block feels familiar but none are exactly the same.

At first, these things seem harmless. A little maintenance task to get to later. Yet over time, this kind of manual sprawl does more than just waste your energy, it introduces fragility into your system

It doesn't have to be that way. There is a better foundation waiting just beneath the surface. And it starts by making the right structure the easy default.

Let's fix this.

The Problem

You're probably using a wildcard certificate already. That's a good start. But if NGINX still treats every subdomain as a separate special case, you haven't really solved the problem.

Because that just means you've managed to centralize just one part of the bigger picture: the certificate. Not so much the logic surrounding how it is used.

The Pattern

Solving this is surprisingly straightforward. You create a single redirect that catches all HTTP traffic, no matter which subdomain it comes from. You pull your SSL parameters into a shared file so that every subdomain inherits the same secure defaults. And instead of bloated, repetitive

The Invisible CDN Trick Behind Every Display Pic

We often think of CDNs as third-party tools or platforms you pay for, with dashboards and edge locations and lots of boxes ticked behind the scenes.

But for one specific use case, long before anyone was throwing around terms like "edge compute" or "dynamic delivery," we had already solved it but not with a service, a pattern.

What we built didn't look like a CDN. It had no domain change and no purge strategy.

But it behaved like one. It delivered dynamic content at static speed. And more than a decade later, the pattern still holds up.

This section is about the trick behind it: serving dynamic assets through NGINX using internal redirects.

No plugin. No third-party API. Just your own server used well.

The Problem

Let's say you run a social app. Each user has a profile picture. Sometimes called an avatar or DP, depending on which corner of the internet you came from.

DPs sound simple but they introduce subtle complications. When a user updates their avatar, that new image needs to appear instantly (across old posts, messages and profile views etc.) without sacrificing speed or breaking the cache.

If you serve images through a backend script, they can't be cached effectively. If you generate new filenames for each update, old content becomes outdated and if you overwrite existing files, you risk stale caches that are hard to control.

Epilogue

Thank you for reading until the end. If you've followed the examples, your NGINX setup should already feel lighter to maintain. You might even notice the difference in real time, especially in how your stack behaves under load.

This volume deliberately focused on what can be achieved with mostly static configurations. But even these benefit from ongoing observation. Analysing traffic patterns over time, reviewing your caching behaviour and paying attention to bottlenecks can reveal small tweaks that lead to major improvements.

In other words, while these configurations are *reactive* by nature, they should not be treated as fire-and-forget. A little strategic intervention now and then can make even a static setup adapt to changing needs.

Volume 2 will take this foundation further. We'll look at dynamic configurations, scripting with `njs`, and techniques for working with live metadata and runtime decisions. We'll explore how NGINX can integrate into larger ecosystems, interacting with frameworks and tools without losing its simplicity. Most of all, we'll uncover ways to solve common infrastructure problems without relying on expensive external platforms.

All code examples, future updates, and companion material live at

https://geekist.co/nginx-by-example.

That's also where Volume 2 will be released.

If you want to stay ahead of that release, the best place is geekist.co. It's where I write about all things Geek.

If this book helped you and you'd like to recommend it to others, you can join the partner programme. Affiliate details are available on the store page. If you'd like to support the work behind this book, or the journey that made it necessary, you can do so at paypal.me/jjnathan.

I'm grateful either way, this book already means more than you know.