About HTTPS

It is easy to assume that HTTPS is something that is just "enabled" or not.

But it is way more complex than that.

!!! tip If you are in a hurry or don't care, continue with the next sections for step by step instructions to set everything up with different techniques.

To **learn the basics of HTTPS**, from a consumer perspective, check https://howhttps.works/.

Now, from a **developer's perspective**, here are several things to have in mind while thinking about HTTPS:

- For HTTPS, **the server** needs to **have "certificates"** generated by a **third party**.
  - Those certificates are actually **acquired** from the third party, not "generated".
- Certificates have a **lifetime**.
  - They **expire**.
  - And then they need to be **renewed**, **acquired again** from the third party.
- The encryption of the connection happens at the **TCP level**.
  - That's one layer **below HTTP**.
  - So, the **certificate and encryption** handling is done **before HTTP**.
- **TCP doesn't know about "domains"**. Only about IP addresses.
  - The information about the **specific domain** requested goes in the **HTTP data**.
- The **HTTPS certificates** "certify" a **certain domain**, but the protocol and encryption happen at the TCP level, **before knowing** which domain is being dealt with.
- **By default**, that would mean that you can only have **one HTTPS certificate per IP address**.
  - No matter how big your server is or how small each application you have on it might be.
  - There is a **solution** to this, however.
- There's an **extension** to the **TLS** protocol (the one handling the encryption at the TCP level, before HTTP) called **SNI**.
  - This SNI extension allows one single server (with a **single IP address**) to have **several HTTPS certificates** and serve **multiple HTTPS domains/applications**.
  - For this to work, a **single** component (program) running on the server, listening on the **public IP address**, must have **all the HTTPS certificates** in the server.
- **After** obtaining a secure connection, the communication protocol is **still HTTP**.
  - The contents are **encrypted**, even though they are being sent with the **HTTP protocol**.

It is a common practice to have **one program/HTTP server** running on the server (the machine, host, etc.) and **managing all the HTTPS parts**: receiving the **encrypted HTTPS requests**, sending the **decrypted HTTP requests** to the actual HTTP application running in the same server (the **FastAPI** application, in this case), take the **HTTP response** from the application, **encrypt it** using the appropriate **HTTPS certificate** and sending it back to the client using **HTTPS**. This server is often called a **TLS Termination Proxy**.

Some of the options you could use as a TLS Termination Proxy are:

- Traefik (that can also handle certificate renewals)
- Caddy (that can also handle certificate renewals)
- Nginx
- HAProxy

Let's Encrypt

Before Let's Encrypt, these **HTTPS certificates** were sold by trusted third parties.

The process to acquire one of these certificates used to be cumbersome, require quite some paperwork and the certificates were quite expensive.

But then **Let's Encrypt** was created.

It is a project from the Linux Foundation. It provides **HTTPS certificates for free**, in an automated way. These certificates use all the standard cryptographic security, and are short-lived (about 3 months), so the **security is actually better** because of their reduced lifespan.

The domains are securely verified and the certificates are generated automatically. This also allows automating the renewal of these certificates.

The idea is to automate the acquisition and renewal of these certificates so that you can have **secure HTTPS, for free, forever**.

HTTPS for Developers

Here's an example of how an HTTPS API could look like, step by step, paying attention mainly to the ideas important for developers.

Domain Name

It would probably all start by you **acquiring** some **domain name**. Then, you would configure it in a DNS server (possibly your same cloud provider).

You would probably get a cloud server (a virtual machine) or something similar, and it would have a fixed **public IP address**.

In the DNS server(s) you would configure a record (an "`A record`") to point **your domain** to the public **IP address of your server**.

You would probably do this just once, the first time, when setting everything up.

!!! tip This Domain Name part is way before HTTPS, but as everything depends on the domain and the IP address, it's worth mentioning it here.

DNS

Now let's focus on all the actual HTTPS parts.

First, the browser would check with the **DNS servers** what is the **IP for the domain**, in this case, `someapp.example.com`.

The DNS servers would tell the browser to use some specific **IP address**. That would be the public IP address used by your server, that you configured in the DNS servers.

TLS Handshake Start

The browser would then communicate with that IP address on **port 443** (the HTTPS port).

The first part of the communication is just to establish the connection between the client and the server and to decide the cryptographic keys they will use, etc.

This interaction between the client and the server to establish the TLS connection is called the **TLS handshake**.

TLS with SNI Extension

Only one process in the server can be listening on a specific **port** in a specific **IP address**. There could be other processes listening on other ports in the same IP address, but only one for each combination of IP address and port.

TLS (HTTPS) uses the specific port `443` by default. So that's the port we would need.

As only one process can be listening on this port, the process that would do it would be the **TLS Termination Proxy**.

The TLS Termination Proxy would have access to one or more **TLS certificates** (HTTPS certificates).

Using the **SNI extension** discussed above, the TLS Termination Proxy would check which of the TLS (HTTPS) certificates available it should use for this connection, using the one that matches the domain expected by the client.

In this case, it would use the certificate for `someapp.example.com`.

The client already **trusts** the entity that generated that TLS certificate (in this case Let's Encrypt, but we'll see about that later), so it can **verify** that the certificate is valid.

Then, using the certificate, the client and the TLS Termination Proxy **decide how to encrypt** the rest of the **TCP communication**. This completes the **TLS Handshake** part.

After this, the client and the server have an **encrypted TCP connection**, this is what TLS provides. And then they can use that connection to start the actual **HTTP communication**.

And that's what **HTTPS** is, it's just plain **HTTP** inside a **secure TLS connection** instead of a pure (unencrypted) TCP connection.

!!! tip Notice that the encryption of the communication happens at the **TCP level**, not at the HTTP level.

HTTPS Request

Now that the client and server (specifically the browser and the TLS Termination Proxy) have an **encrypted TCP connection**, they can start the **HTTP communication**.

So, the client sends an **HTTPS request**. This is just an HTTP request through an encrypted TLS connection.

Decrypt the Request

The TLS Termination Proxy would use the encryption agreed to **decrypt the request**, and would transmit the **plain (decrypted) HTTP request** to the process running the application (for example a process with Uvicorn running the FastAPI application).

HTTP Response

The application would process the request and send a **plain (unencrypted) HTTP response** to the TLS Termination Proxy.

HTTPS Response

The TLS Termination Proxy would then **encrypt the response** using the cryptography agreed before (that started with the certificate for `someapp.example.com`), and send it back to the browser.

Next, the browser would verify that the response is valid and encrypted with the right cryptographic key, etc. It would then **decrypt the response** and process it.

The client (browser) will know that the response comes from the correct server because it is using the cryptography they agreed using the **HTTPS certificate** before.

Multiple Applications

In the same server (or servers), there could be **multiple applications**, for example, other API programs or a database.

Only one process can be handling the specific IP and port (the TLS Termination Proxy in our example) but the other applications/processes can be running on the server(s) too, as long as they don't try to use the same **combination of public IP and port**.

That way, the TLS Termination Proxy could handle HTTPS and certificates for **multiple domains**, for multiple applications, and then transmit the requests to the right application in each case.

Certificate Renewal

At some point in the future, each certificate would **expire** (about 3 months after acquiring it).

And then, there would be another program (in some cases it's another program, in some cases it could be the same TLS Termination Proxy) that would talk to Let's Encrypt, and renew the certificate(s).

The **TLS certificates** are **associated with a domain name**, not with an IP address.

So, to renew the certificates, the renewal program needs to **prove** to the authority (Let's Encrypt) that it indeed **"owns" and controls that domain**.

To do that, and to accommodate different application needs, there are several ways it can do it. Some popular ways are:

- **Modify some DNS records**.
  - For this, the renewal program needs to support the APIs of the DNS provider, so, depending on the DNS provider you are using, this might or might not be an option.
- **Run as a server** (at least during the certificate acquisition process) on the public IP address associated with the domain.

    - As we said above, only one process can be listening on a specific IP and port.
    - This is one of the reasons why it's very useful when the same TLS Termination Proxy also takes care of the certificate renewal process.
    - Otherwise, you might have to stop the TLS Termination Proxy momentarily, start the renewal program to acquire the certificates, then configure them with the TLS Termination Proxy, and then restart the TLS Termination Proxy. This is not ideal, as your app(s) will not be available during the time that the TLS Termination Proxy is off.

All this renewal process, while still serving the app, is one of the main reasons why you would want to have a **separate system to handle HTTPS** with a TLS Termination Proxy instead of just using the TLS certificates with the application server directly (e.g. Uvicorn).

Recap

Having **HTTPS** is very important, and quite **critical** in most cases. Most of the effort you as a developer have to put around HTTPS is just about **understanding these concepts** and how they work.

But once you know the basic information of **HTTPS for developers** you can easily combine and configure different tools to help you manage everything in a simple way.

In some of the next chapters, I'll show you several concrete examples of how to set up **HTTPS** for **FastAPI** applications. 🔒