What Happens When You Type a URL and Press Enter

A complete, plain-language study guide

NetSec Visualized · Companion to Episode 001 · Aligned to Network+, Security+, and CCNA

This guide walks through everything that happens between pressing Enter and a web page appearing on your screen. It follows one request from your keyboard to the pixels, and explains each piece of machinery it touches. Every term is defined the first time it appears. You do not need a networking background to read it.

This is the deeper companion to the video. The main text explains each idea in plain language; the **Go deeper** notes add the detail an engineer or a certification exam expects; and the **Common misconception** boxes clear up the things people most often get wrong. Use the **What to remember** boxes and the glossary to review.

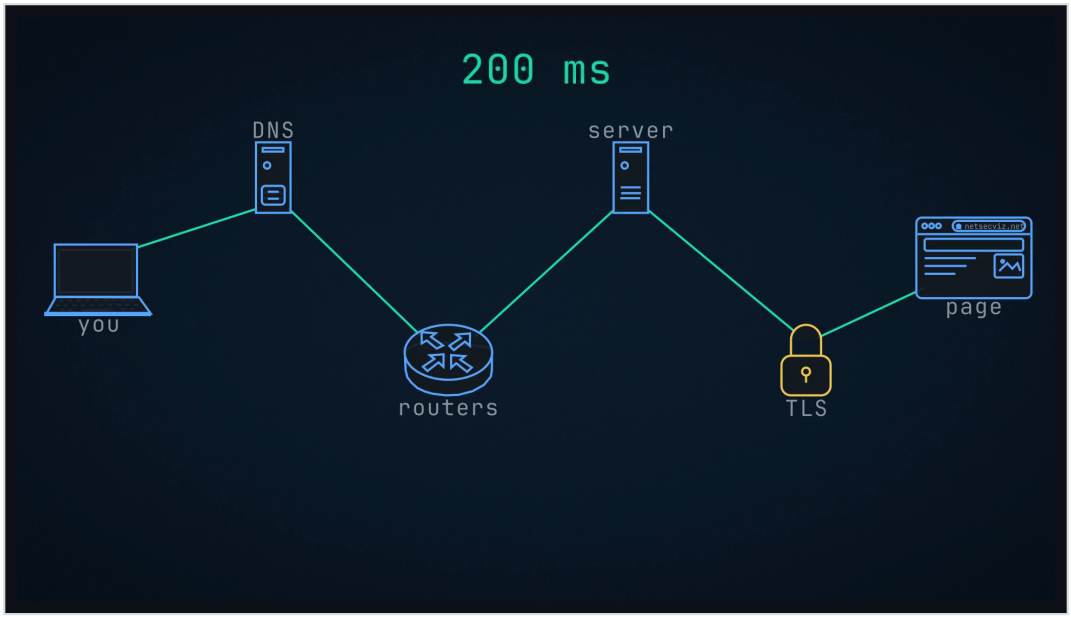
The big picture

When you load a web page, your computer runs through several stages in well under a second:

1. **The URL** is parsed into pieces, and the connection is upgraded to a secure one if required.
2. **DNS** turns the human name (like `netsecviz.net`) into a numeric IP address.
3. **ARP** supplies the hardware address of your router so data can leave your local network. (Usually this is already cached, more on that below.)
4. **TCP** opens a reliable connection to the server with a three-way handshake.
5. **TLS** turns that connection into a private, encrypted one.
6. **HTTP** is the actual request for the page, and the server's response.
7. **Rendering** turns the returned code into the pixels you see.

A newer path, **HTTP/3 over QUIC**, folds stages 4 and 5 together to save time. We cover it after HTTP.

One honest note up front: a real page load is not perfectly linear. Caches mean whole stages are often skipped, and the browser does many things in parallel. The order below is the logical journey, which is the right way to learn it. Where reality diverges, the notes say so.



the full journey, from your laptop through DNS, routers, the server, and TLS, to the rendered page.

Stage 1 — The URL and the keystroke

In one sentence: *before anything leaves your computer, the browser reads the address you typed and decides exactly what you are asking for.*

A **URL** (Uniform Resource Locator) is the full address of a resource on the web. The browser splits it into parts:

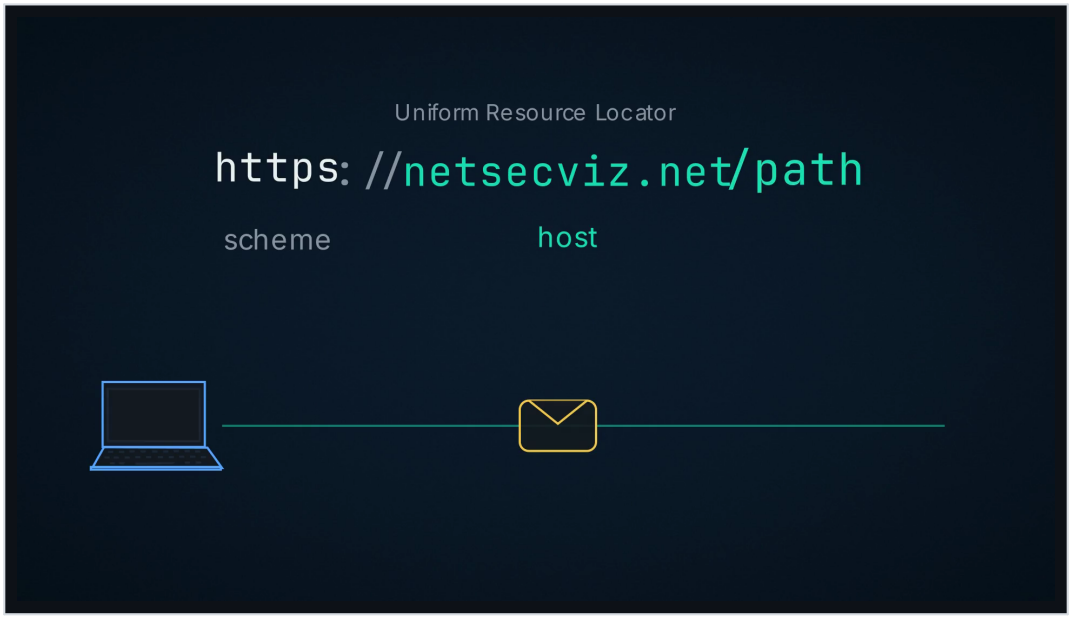
- **Scheme** (`https`): the protocol to use. HTTPS means HTTP carried over an encrypted connection.
- **Host** (`netsecviz.net`): the name of the machine you want to reach.
- **Path** (`/path`): which specific resource on that host.

The browser also decides whether what you typed is an address at all or a search term. If the site is on the browser's **HSTS** list (HTTP Strict Transport Security, a list of sites that must always be used securely), the browser silently upgrades `http` to `https` before sending anything. This prevents an attacker from downgrading you to an unencrypted connection.

> **GO DEEPER**
>
> a full URL can also carry a port (`:443`), a query string (`?id=42`), and a fragment (`#section`). The fragment never leaves the browser; it is used locally to scroll to part of the page. Browsers also ship with a preloaded HSTS list, so even a site you have never visited can be forced to HTTPS on the very first request.

> **WHAT TO REMEMBER**

> a URL is scheme + host + path (with optional port, query, and fragment). HTTPS is HTTP plus encryption. HSTS forces a secure connection, even on a first visit via the preload list.



a URL broken into scheme, host, and path.

Stage 2 — DNS: turning a name into a number

In one sentence: *computers route by numbers, not names, so the browser first looks up the IP address for the host.*

An **IP address** (for example `203.0.113.42`) is the numeric address of a machine on a network. **DNS** (the Domain Name System) is the system that maps names to those numbers. Think of it as the phone book of the internet.

The browser does not ask the internet right away. It checks **caches** first, in order, because a recent answer is faster than a fresh lookup:

1. The **browser cache** (its own memory of recent lookups).
2. The **operating system cache** (the OS keeps its own copy, via the stub resolver).

On a miss, it asks a **recursive resolver**, a server whose job is to do the legwork for you (often your ISP's, or a public one like `8.8.8.8`). If the resolver has no cached answer, it walks the DNS hierarchy:

1. A **root server** does not know the answer but knows who handles `.net`, and points to the right TLD server.
2. A **TLD server** (Top-Level Domain server, here the one for `.net`) knows which server is authoritative for `netsecviz.net`.
3. The **authoritative server** holds the real record and returns the actual IP address.

That IP travels back to your machine. Every answer carries a **TTL** (Time To Live), an expiry timer that says how long it may be cached.

GO DEEPER

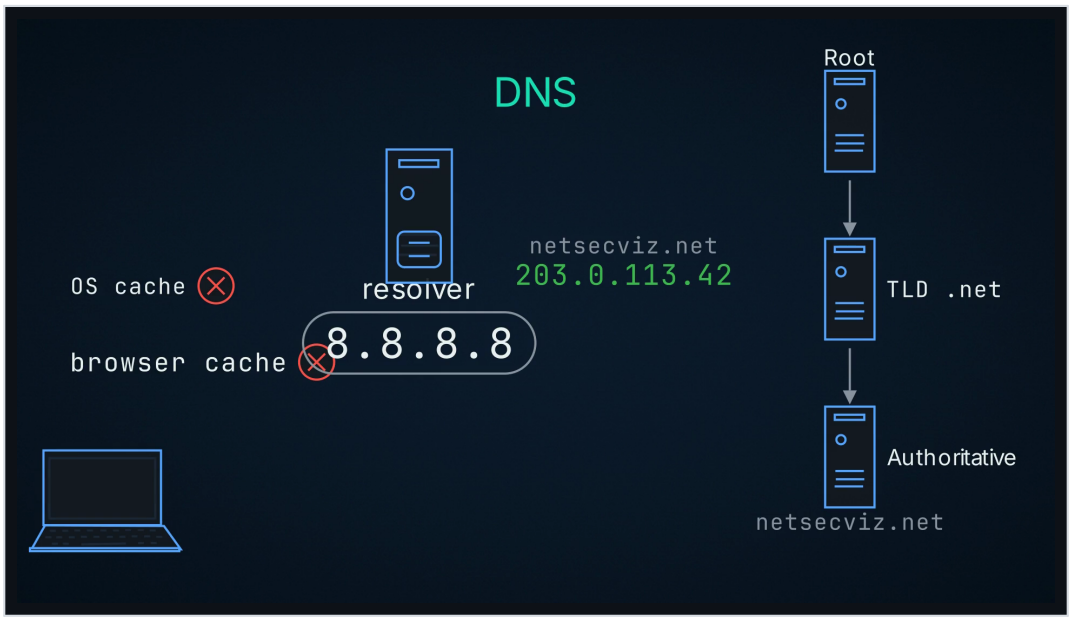
DNS answers come as **records** of different types. An **A record** maps a name to an IPv4 address; an **AAAA record** maps it to an IPv6 address; a **CNAME** points one name at another name. The resolver's walk down the hierarchy is **iterative** (each server refers it to the next), while the service the resolver provides to you is **recursive** (it chases the whole chain on your behalf). Increasingly, DNS itself is encrypted with **DNS over HTTPS (DoH)** or DNS over TLS, so the lookup is private too.

COMMON MISCONCEPTION

DNS is not queried fresh on every page load. Because of caching at the browser, OS, and resolver, the full root-to-authoritative walk is the exception, not the rule. Most lookups are answered from a cache two steps away and never leave your machine or your resolver.

WHAT TO REMEMBER

resolution order is browser cache → OS cache → recursive resolver → root → TLD → authoritative. A **recursive** resolver does the full lookup for you; an **authoritative** server holds the actual record. A and AAAA records hold IPv4 and IPv6 addresses. TTL controls caching.



the resolver walking root → TLD → authoritative and returning the IP, with caches on the left.

Stage 3 — ARP: getting off your own network

In one sentence: *to send data beyond your local network, your computer hands it to your router, and* ***ARP*** *is how it learns the router's hardware address, normally just once, then remembers it.*

Your machine now has the server's IP address, but it cannot reach it directly. It can only physically deliver data to devices on its own **local network (LAN)**. Anything outside the LAN has to go through the **default gateway**, which is your router, the one device with a door to the outside world.

To hand data to the router, your computer needs the router's **MAC address** (Media Access Control address, a hardware serial number on the network card). **ARP** (Address Resolution Protocol) is how it finds that out: it broadcasts a question to the local network, "who has `192.168.1.1`? Tell me your MAC," and the router replies with its MAC.

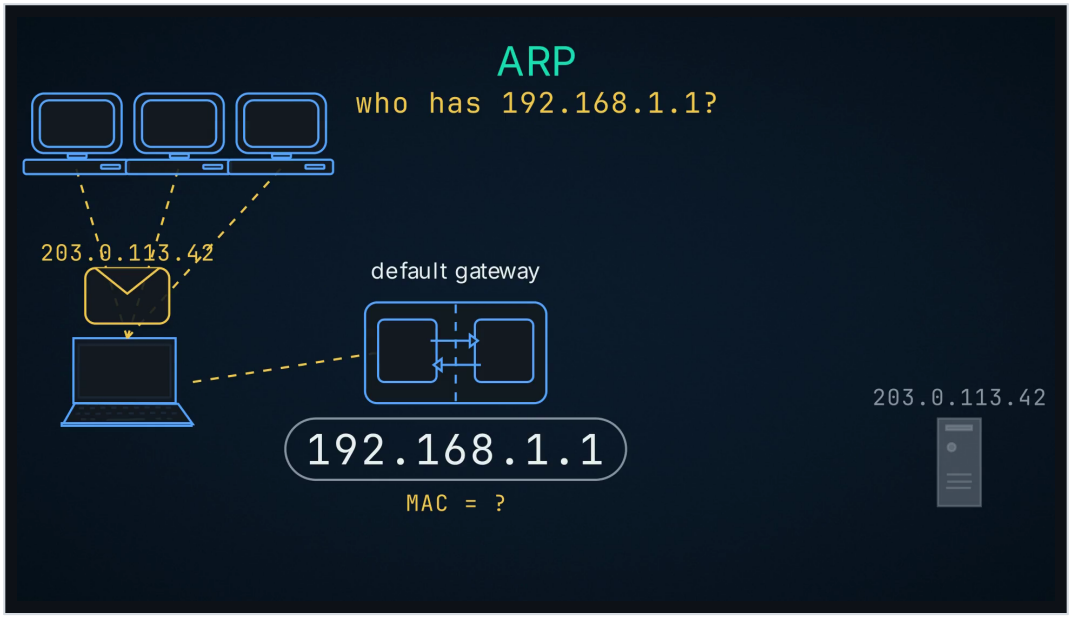
Here is the part the video did not have time for. Your computer stores what it learns in an **ARP cache** (also called the ARP table), a small list of recent IP-to-MAC mappings. ARP only sends that broadcast when it needs a mapping it does not already have. The default gateway's MAC is typically resolved **once**, the first time your machine needs to talk to anything off the local network after joining the network, and then it is reused for every connection after that. It is refreshed only when the cache entry expires (commonly minutes) or the device drops off the network. So in the journey of a single web request, ARP has usually **already happened**; we show it because it is essential to understand, not because it fires on this particular request.

Once the gateway's MAC is known, the data travels **hop by hop**: to your router, then the next router, then the next, across the internet to the server. At each hop the destination IP address stays the same, but the hardware (MAC) address changes to that of the next device in line. (Each of those later hops does its own address resolution on its own network; you do not.)

> **COMMON MISCONCEPTION**
>
> ARP does not run on every request, every connection, or every packet. The gateway's MAC is cached after the first resolution and reused. ARP repeats only when the cache entry times out or the machine rejoins the network. Framing ARP as "happens every time you make a network call" is the mistake to avoid.

> **GO DEEPER**
>
> you can see this cache yourself with `arp -a` (Windows and macOS) or `ip neigh` (Linux). Devices also send a **gratuitous ARP** when they join a network, announcing their own IP-to-MAC mapping so neighbors can pre-populate their tables. ARP operates only within a single local network (Layer 2); it never crosses a router. This local-only, trust-by-default design is also why **ARP spoofing** is possible, a topic for a later video.

> **WHAT TO REMEMBER**
>
> off-network traffic goes to the default gateway. ARP maps an IP to a MAC on the local network and the result is cached, so it is essentially a one-time step per gateway, not per request. Across the internet, the IP stays constant end to end while the MAC changes at every hop.



an ARP broadcast asking "who has 192.168.1.1?" with the gateway answering.

Stage 4 — TCP: the three-way handshake

In one sentence: *before sending real data, both sides agree to talk using a short, reliable handshake.*

TCP (Transmission Control Protocol) is the protocol that makes a connection **reliable**: data arrives in order, and anything lost is re-sent. A connection targets the server's IP on a specific **port** (port 443 for HTTPS). Ports let one machine run many services at once.

TCP opens with a **three-way handshake**:

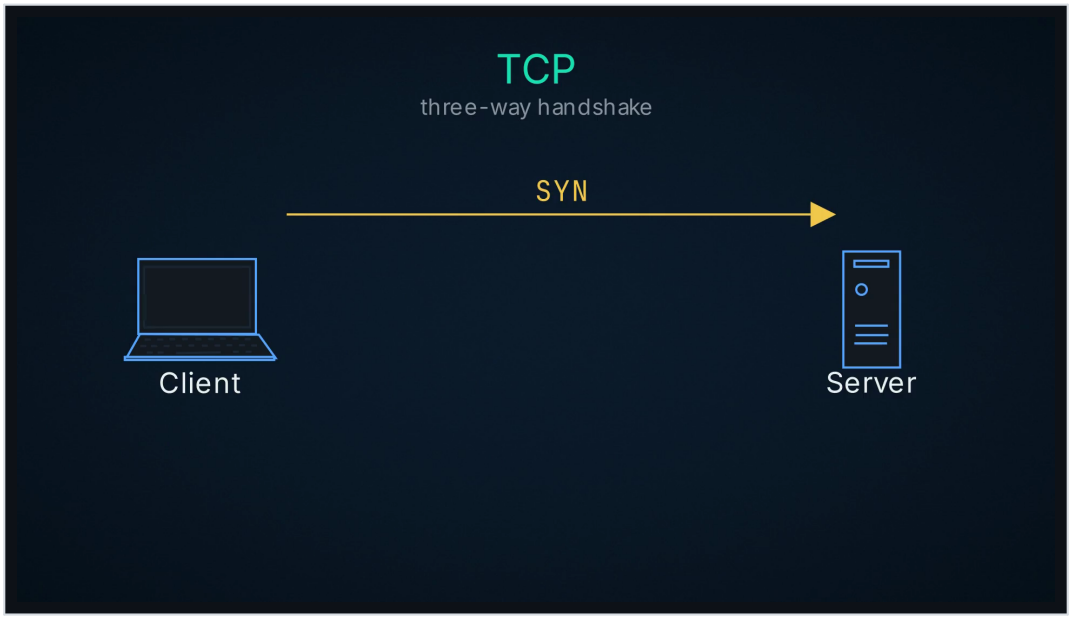
1. Your browser sends a **SYN** (synchronize) to ask to start.
2. The server replies with a **SYN-ACK** (synchronize + acknowledge) to agree.
3. Your browser sends an **ACK** (acknowledge) to confirm.

Three small packets, and now there is an open, reliable pipe between the two machines. But an open pipe is not yet a private one, which is what the next stage fixes.

> **GO DEEPER**
>
> the handshake does more than say hello. Each side picks a random starting **sequence number** and tells the other, so that from then on every byte is numbered, lost bytes can be re-sent, and out-of-order bytes can be reassembled. That is why it takes three messages and not two: both sides must announce their sequence number and have it acknowledged. The combination of an IP address and a port is called a **socket**, and a connection is uniquely identified by the pair of sockets at each end.

> **WHAT TO REMEMBER**

> the handshake is SYN → SYN-ACK → ACK; it exists to synchronize sequence numbers and confirm both directions work. TCP guarantees ordered, reliable delivery. HTTPS uses port 443. An IP plus a port is a socket.



the SYN going from client to server at the start of the handshake.

Stage 5 — TLS: making it private

In one sentence: *anyone sitting on the wire can read plain TCP traffic, so TLS encrypts the connection before any real data is sent.*

TLS (Transport Layer Security) is the protocol that encrypts the connection. It is the **S in HTTPS**, and the closed padlock in your browser means TLS is active. Modern sites use **TLS 1.3**, which sets up in a single round trip.

The naive way to encrypt would be to pick a secret key and send it to the server, but anyone listening would just grab the key in transit. TLS avoids this entirely:

1. Your browser sends a **ClientHello** that includes its half of a **Diffie-Hellman** key share. (Diffie-Hellman is a method where two parties each generate a public and a private half.)
2. The server replies with a **ServerHello** that includes its own public half, plus its **certificate**.
3. The browser checks the **certificate**, a digital ID card signed by a trusted **Certificate Authority (CA)**, to confirm it is really talking to `netsecviz.net` and not an imposter.
4. Each side mathematically combines its own private half with the other side's public half. Both arrive at the **exact same shared secret**, a secret that was never sent across the wire.

From this point on, everything is encrypted with that shared key.

> **GO DEEPER**

the key exchange uses **asymmetric** cryptography (separate public and private halves) only to agree on a key. The actual page data is then encrypted with fast **symmetric** cryptography (one shared key for both sides), because symmetric encryption is far quicker for bulk data. The certificate also enables **authentication**: the CA's signature is what proves identity, while Diffie-Hellman provides the secrecy. Because one server IP can host many sites, the browser sends the hostname it wants in a field called **SNI** (Server Name Indication) so the server presents the right certificate.

COMMON MISCONCEPTION

the certificate's key is not the key that encrypts your traffic. The certificate proves identity and helps authenticate the exchange; the actual session key is derived freshly by both sides via Diffie-Hellman and is never transmitted. This separation is exactly what gives modern TLS its **forward secrecy**: even if an attacker records all of today's encrypted traffic and later steals the server's private key, they still cannot decrypt what they recorded, because the session key existed only for that one conversation and then vanished.

WHAT TO REMEMBER

TLS encrypts the TCP connection. Asymmetric crypto (Diffie-Hellman) agrees on a key; symmetric crypto encrypts the data. The certificate proves identity; the session key is never sent. Forward secrecy keeps past traffic safe even if the server key later leaks.

the Diffie-Hellman exchange and certificate check; the padlock closing once the secret is agreed.

Stage 6 — HTTP: asking for the page

In one sentence: *through the now-encrypted pipe, the browser sends a small text request and the server sends back the page.*

HTTP (HyperText Transfer Protocol) is the language of the request itself, and it is surprisingly small. A request is often just one line plus a few headers:

- The **method and path**: `GET /` means "give me the resource at the root path." GET is the method for retrieving something.

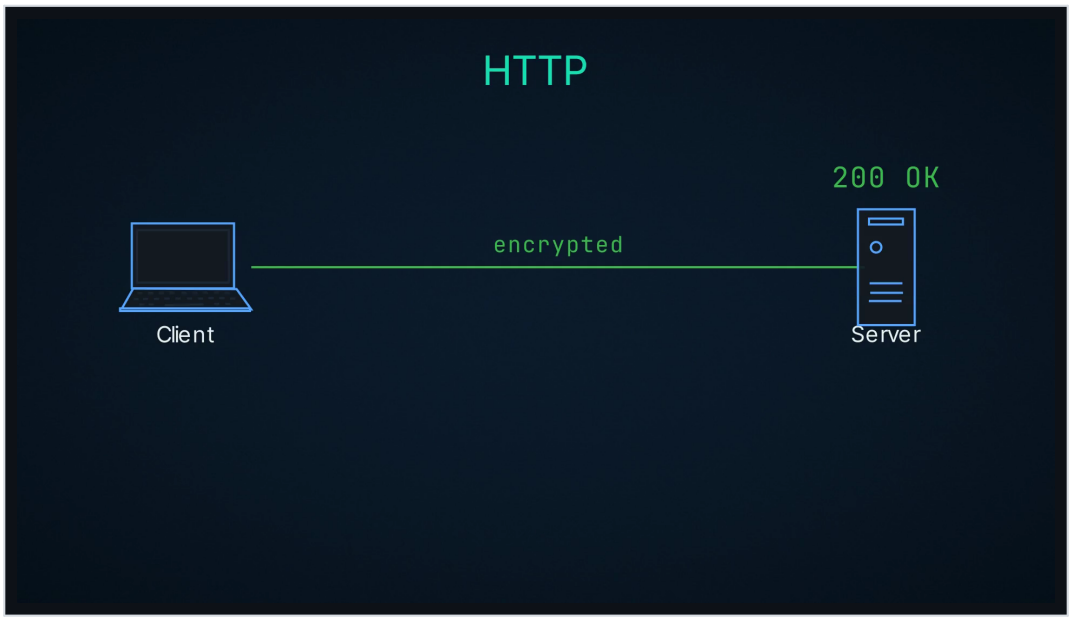
- **Headers**: extra key-value details, such as `Host:` (which site you want) and what content types you accept.

The server reads the request, does its work, and answers with:

- A **status code**: `200 OK` means success.
- Response **headers**.
- The **body**: the actual HTML document.

> **GO DEEPER**
>
> common methods are **GET** (retrieve), **POST** (send data), **PUT** (replace), and **DELETE** (remove). Status codes come in families: **2xx** success (200 OK), **3xx** redirects (301 moved permanently, 302 found), **4xx** client errors (404 not found, 403 forbidden), and **5xx** server errors (500 internal error, 503 unavailable). Headers do real work: `Cache-Control` decides what the browser may reuse, `Content-Type` says what the body is, and `Set-Cookie` stores state. The `Host` header is what lets one server host many sites on one IP.

> **WHAT TO REMEMBER**
>
> HTTP is a request (method + path + headers) and a response (status code + headers + body). 2xx is success, 3xx redirect, 4xx your error, 5xx the server's error. The body of the first response is the HTML.



the server returning a 200 OK with the HTML body.

Stage 7 — HTTP/3 and QUIC: the shortcut

In one sentence: *the newest version of the web combines the TCP and TLS handshakes into one to load pages faster.*

In the classic path, we did two handshakes back to back: TCP to make the connection reliable, then TLS to make it private. That is two full round trips before any page data moves.

HTTP/3 is the newest version of HTTP, and it runs on a protocol called **QUIC** instead of TCP. QUIC runs on **UDP** (User Datagram Protocol), a leaner, faster way of sending packets, and it folds the reliability and encryption setup into a single step. That collapses the setup from two round trips down to one.

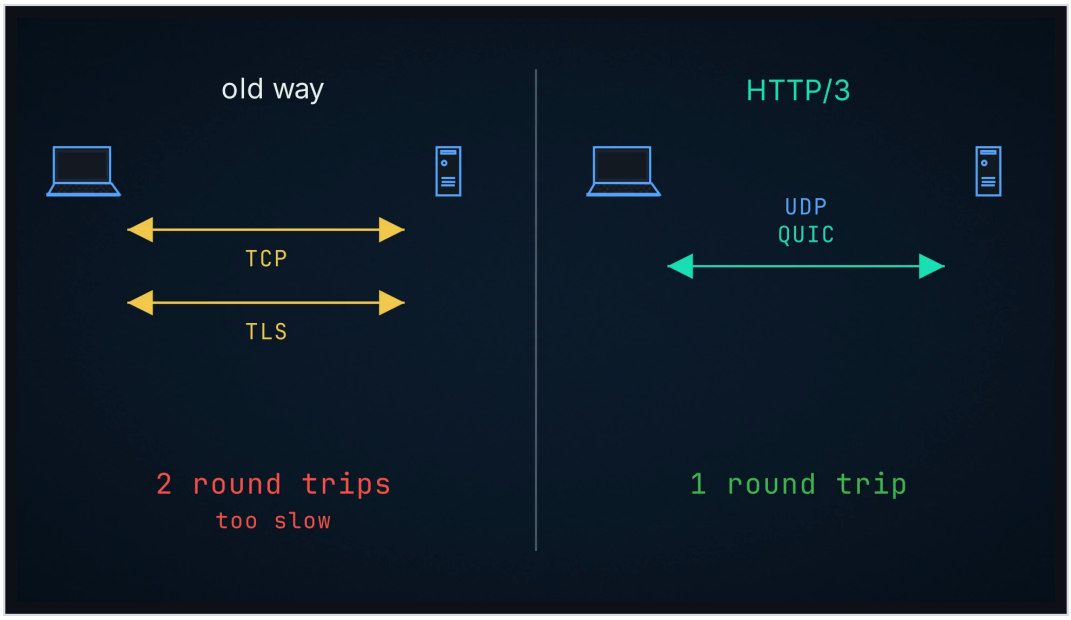
For repeat visitors it goes further. A feature called **0-RTT** (zero round-trip time) lets a returning browser send encrypted data in its very first packet, with no handshake at all.

GO DEEPER

UDP by itself is unreliable (it does not re-send lost packets), so QUIC rebuilds reliability and ordering on top of UDP, while keeping the freedom to innovate that a brand-new protocol allows. A big practical win is the end of **head-of-line blocking**: in older HTTP over one TCP connection, one lost packet stalled every stream behind it; QUIC keeps streams independent, so a single loss only pauses its own stream. The trade-off worth knowing: 0-RTT data can be replayed by an attacker, so it is only safe for requests that do not change state (like a GET).

WHAT TO REMEMBER

HTTP/3 runs on QUIC over UDP and merges the TCP and TLS handshakes into one round trip. 0-RTT lets returning visitors skip the handshake. QUIC removes head-of-line blocking by keeping streams independent.



the round-trip count dropping as QUIC merges the handshakes.

Stage 8 — Rendering: from text to pixels

In one sentence: *the document the server sent is just text, so the browser runs a pipeline to turn it into the picture on your screen.*

The browser receives HTML, which is just text, and your screen needs pixels. It runs a **render pipeline**:

1. **Parse HTML into the DOM** (Document Object Model), a tree of every element on the page.
2. **Parse CSS into the CSSOM** (CSS Object Model), a matching tree of all the styling rules.
3. **Combine them into the render tree**, which holds only the things that are actually visible.
4. **Layout**: work out the position and size of every element (the geometry).
5. **Paint**: fill in the actual pixels, colors, text, and borders.
6. **Composite**: hand the layers to the **GPU** (graphics processor) to assemble the final image efficiently.

The browser does not wait for everything. It **renders progressively**, painting what it can as fast as it can, and fires off more requests for images, fonts, and scripts. Each of those is its own small version of this entire journey.

GO DEEPER

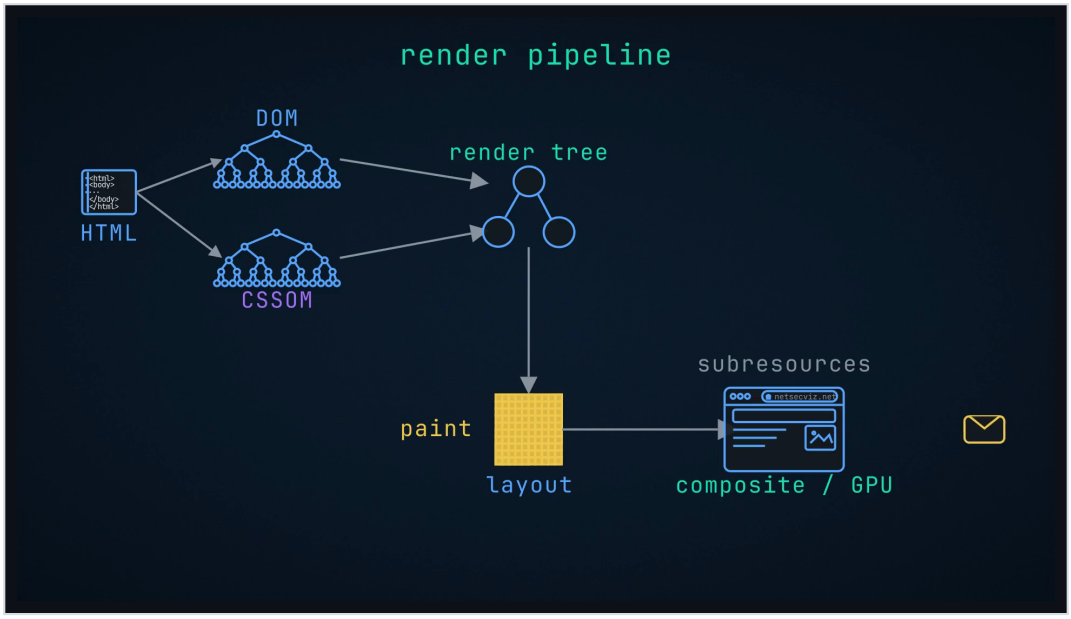
this whole sequence is called the **critical rendering path**, and shortening it is the heart of web performance. Two terms you will hear: **reflow** (the browser has to recompute layout, because something changed size or position) and **repaint** (it only has to redraw pixels, like a color change). Reflows are more expensive. CSS is treated as **render-blocking** because the browser needs the CSSOM before it can build the render tree, while scripts can block parsing unless marked `async` or `defer`.

COMMON MISCONCEPTION

the render tree is not the same as the DOM. The DOM contains every element, but the render tree contains only what is visible; elements hidden with `display: none`, and non-visual tags like `<head>`, are in the DOM but excluded from the render tree.

WHAT TO REMEMBER

the pipeline is DOM + CSSOM → render tree → layout → paint → composite (the critical rendering path). The render tree holds only visible nodes. Reflow recomputes layout; repaint only redraws. Pages render progressively.



the render pipeline from HTML and CSS through to the composited page.

Glossary

- **0-RTT (zero round-trip time):** a QUIC feature letting a returning client send encrypted data in the first packet, skipping the handshake.
- **A / AAAA record:** DNS records mapping a name to an IPv4 (A) or IPv6 (AAAA) address.
- **ACK:** acknowledge; the third message of the TCP handshake.
- **ARP (Address Resolution Protocol):** maps an IP address to a MAC address on the local network; results are cached.
- **ARP cache (ARP table):** the list of recently learned IP-to-MAC mappings a device keeps so it does not re-ARP every time.
- **Asymmetric / symmetric crypto:** asymmetric uses a public and private key pair (used to agree on a key); symmetric uses one shared key (used to encrypt the data).
- **Authoritative server:** the DNS server that holds the real record for a domain.
- **Cache:** a stored copy of a recent answer, used to skip repeated work.
- **Certificate:** a digital ID proving a server's identity, signed by a Certificate Authority.
- **Certificate Authority (CA):** a trusted organization that signs certificates.
- **CNAME:** a DNS record that points one name to another name.
- **Composite:** the final render step where the GPU assembles layers into the image.
- **Critical rendering path:** the full sequence from HTML and CSS to pixels; shortening it improves load speed.
- **CSSOM (CSS Object Model):** a tree of all CSS styling rules.
- **Default gateway:** the router that connects your local network to the outside world.
- **Diffie-Hellman:** a method letting two parties derive a shared secret without sending it.
- **DNS (Domain Name System):** the system that maps names to IP addresses.
- **DNS over HTTPS (DoH):** DNS lookups sent over an encrypted HTTPS connection for privacy.
- **DOM (Document Object Model):** a tree of every element on a page.
- **Forward secrecy:** a property where past encrypted traffic stays safe even if the server's key later leaks.
- **Gratuitous ARP:** an unsolicited ARP a device sends when joining a network to announce its own mapping.
- **GPU (Graphics Processing Unit):** the processor that composites and draws the final image.
- **Head-of-line blocking:** when one lost packet stalls everything behind it; QUIC avoids it by keeping streams independent.
- **Host:** the machine name portion of a URL.
- **HSTS (HTTP Strict Transport Security):** forces a site to always be used over HTTPS.
- **HTTP (HyperText Transfer Protocol):** the request/response language of the web.
- **HTTP/3:** the newest HTTP version, running on QUIC.
- **IP address:** the numeric address of a machine on a network.

- **Iterative vs recursive (DNS):** the resolver's walk is iterative (referrals); the service it gives you is recursive (it chases the whole chain).
- **LAN (Local Area Network):** your immediate local network.
- **Layout:** the render step that computes element positions and sizes.
- **MAC address (Media Access Control):** a hardware address burned into a network card.
- **Method (GET, POST, ...):** the kind of HTTP request; GET retrieves, POST sends data.
- **Paint:** the render step that fills in pixels and colors.
- **Path:** the part of a URL after the host, naming a specific resource.
- **Port:** a number identifying a service on a machine (443 for HTTPS).
- **QUIC:** a transport protocol over UDP that merges reliability and encryption setup.
- **Recursive resolver:** a DNS server that does the full lookup on your behalf.
- **Reflow / repaint:** reflow recomputes layout; repaint only redraws pixels.
- **Render tree:** the combined DOM + CSSOM, containing only visible elements.
- **Root server:** the top of the DNS hierarchy; points to the right TLD server.
- **Scheme:** the protocol part of a URL (for example `https`).
- **Sequence number:** the per-byte counter TCP sets up in the handshake to order and recover data.
- **SNI (Server Name Indication):** the field telling a shared server which site's certificate to present.
- **Socket:** an IP address plus a port; a connection is a pair of sockets.
- **Status code:** the result of an HTTP request (200 OK, 404 not found, and so on).
- **SYN / SYN-ACK:** the first two messages of the TCP handshake.
- **TCP (Transmission Control Protocol):** provides reliable, ordered delivery.
- **TLD (Top-Level Domain) server:** the DNS server for an extension like `.net`.
- **TLS (Transport Layer Security):** encrypts the connection; the S in HTTPS.
- **TTL (Time To Live):** how long a cached DNS answer stays valid.
- **UDP (User Datagram Protocol):** a lightweight, fast transport that QUIC builds on.
- **URL (Uniform Resource Locator):** the full address of a web resource.

Quick self-test

1. Put the DNS lookup locations in the order they are checked, and name the record type that holds an IPv4 address.

2. When does ARP actually send a broadcast, and why is it wrong to say it happens on every request?

3. List the three messages of the TCP handshake, and explain why two would not be enough.

4. In TLS, which part provides secrecy and which part provides identity? Why can a wiretapper who records the whole handshake still not read the traffic?

5. Match the status code families 2xx, 3xx, 4xx, 5xx to their meanings.

6. How does HTTP/3 over QUIC save time, and what problem does keeping streams independent solve?

7. Name the render pipeline steps, and explain why an element with `display: none` is in the DOM but not the render tree.

(Answers are in the "What to remember" and "Common misconception" boxes above.)

Further reading

The facts in this guide trace to the episode's source list (`01-research/sources.md`), including Cloudflare Learning (DNS), MDN and web.dev (rendering), and Gcore and Sectigo (TLS 1.3). For exam prep, map each stage to its objective: DNS, ARP, TCP, ports, and IP addressing sit in Network+ Networking Fundamentals; TLS, certificates, asymmetric vs symmetric crypto, and forward secrecy sit in Security+ Architecture and Cryptographic Concepts.

NetSec Visualized — networking and security, made visual.