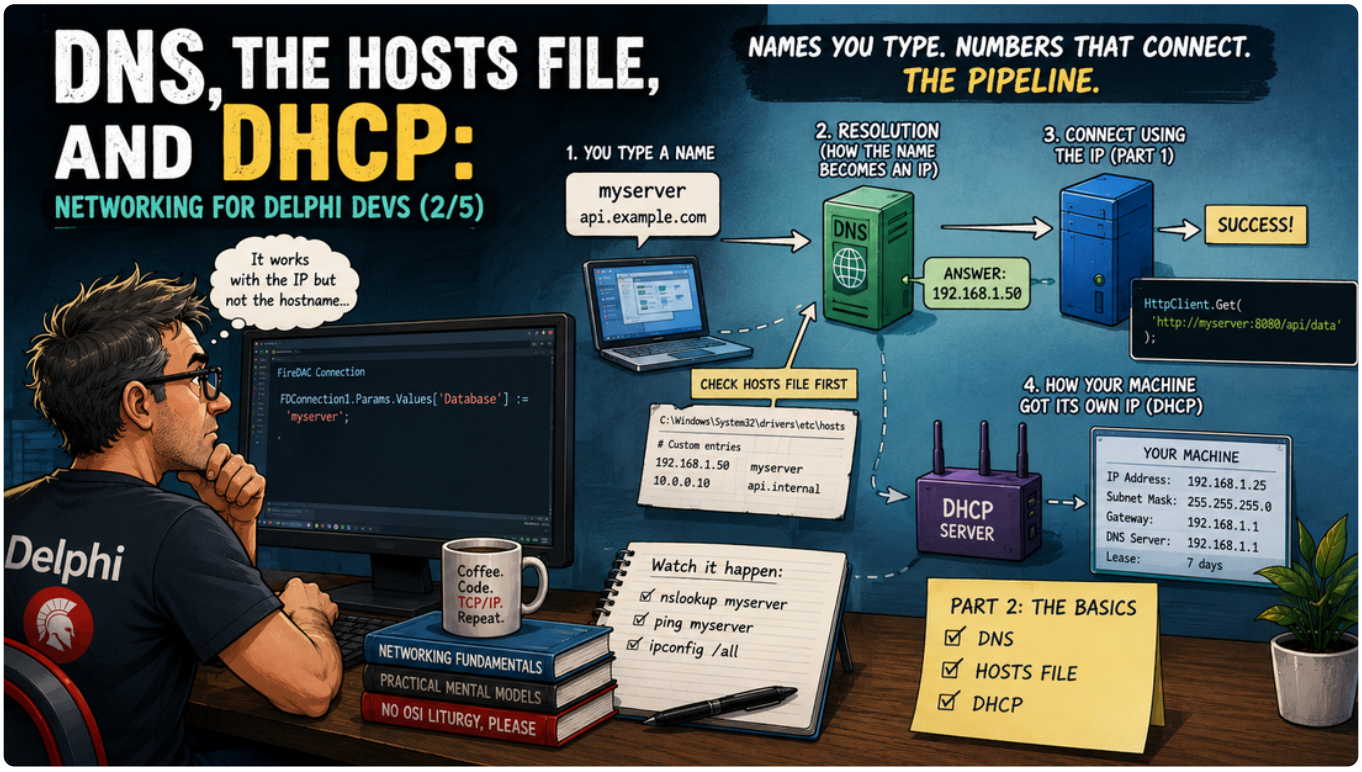
NEWS · JULY 11, 2026

DNS, the hosts File, and DHCP: Networking for Delphi Devs (2/5)

By Dr. Holger Flick



In Part 1 we established the numbers: every machine on a network has an IP address, `127.0.0.1` is always *this* machine, and a port picks the specific program behind that address. Numbers all the way down.

But here's the thing — you almost never *type* those numbers. You type `myserver` into a FireDAC connection string. You point a REST client at `api.example.com`. You open `https://www.embarcadero.com` in a browser. Names everywhere, and somehow they all end up as the numbers from Part 1.

That gap — between the name you type and the connection that opens — is where an entire class of "it works on my machine" bugs lives. *"It works with the IP but not the hostname." "It worked yesterday and the server hasn't changed." "It works on the server itself but not from the client PC."* Every one of those is a name-resolution problem, and every one of them is diagnosable in minutes once you know the pipeline.

This part gives you that pipeline: how a name becomes an address (DNS and the hosts file), and — the other direction — who gave your machine its own address in the first place (DHCP). By the end you'll be able to watch resolution happen on your own machine with three commands.

From a name to a number: the resolution pipeline

When your Delphi app — or any program — asks to connect to `myserver`, it doesn't do the detective work itself. It hands the name to the operating system's **resolver**, a built-in service whose whole job is turning names into IP addresses. The resolver then works through a short checklist, and the first hit wins.

On Windows, the documented order is essentially: check the local cache and the **hosts file** first, then ask a **DNS server**, with older NetBIOS mechanisms as a legacy fallback (Microsoft's host name resolution order).

Here is the whole flow in one picture.

Read that diagram top to bottom on the right side and you have the whole story: the resolver first checks whether it already knows the answer (the cache), then consults a plain text file on your own disk (the hosts file), and only then goes out on the network to ask a DNS server. Whatever answer comes back is cached, handed to your app, and your app connects to the IP exactly as in Part 1. Two things follow immediately, and both matter for debugging:

1. **Your app never talks to DNS directly.** FireDAC, `TNetHTTPClient`, your browser — they all ask the same OS resolver. That's why a name that fails in your Delphi app also fails in the browser on the same machine, and why testing with the browser is a legitimate diagnostic step.
2. **Local answers beat network answers.** A hosts-file entry silently overrides whatever DNS would have said. That's a feature — and occasionally the bug.

The hosts file: your personal phone book override

Before there was DNS, there was literally a shared text file of names and addresses — and a descendant of that file still ships with every operating system. On Windows it lives at `C:\Windows\System32\drivers\etc\hosts` (formally `%WinDir%\System32\drivers\etc`, per Microsoft's hosts-file documentation); it has no file extension, and each line is simply an IP address followed by one or more names:

```
192.168.1.40    myserver
192.168.1.40    api.mycompany.test
```

Because the resolver consults this file *before* asking any DNS server, it's the developer's friend in a very specific way: **you can make any name mean anything, on your machine only, without touching a single server.**

Some everyday uses:

- Point `api.mycompany.test` at a test server's IP so your app's configuration doesn't change between environments — only the hosts file does.
- Point a production hostname at a staging machine to test "the real config" safely from one PC.
- Give a colleague's machine a stable name while you're integrating against it.

> **Editing the hosts file needs administrator rights**
> The file is protected. Open Notepad (or your editor) **as administrator**, then open `C:\Windows\System32\drivers\etc\hosts`. Lines starting with `#` are comments. And remember the flip side of "local answers win": a forgotten hosts entry is a classic source of "this machine behaves differently than every other machine." When resolution looks haunted, check the hosts file first — it takes ten seconds.

DNS: the phone book that nobody owns entirely

For every name that isn't in your hosts file, the resolver asks the Domain Name System — DNS — defined in RFCs 1034 and 1035 and running essentially unchanged in concept since 1987. The mental model that serves you best: **DNS is a distributed phone book.** No single machine holds the whole book; instead, responsibility is delegated. Somebody authoritative for `example.com` publishes the entries for names under it, somebody else is authoritative for `embarcadero.com`, and a hierarchy of servers knows who to ask for what.

The entry type you care about ninety percent of the time is the **A record**: a simple mapping of *name* ’*IPv4 address*, e.g. `api.example.com ’ 93.184.216.34`. Its sibling, the **AAAA record**, does the same for IPv6. There are many other record types — mail routing, aliases, text metadata — registered with IANA, the organization that coordinates the internet's global identifiers. You don't need to learn them now; you need to know they exist so the terms don't scare you when a server admin mentions a "CNAME."

And which DNS server does *your* machine ask? Almost always one it was told about automatically: on a home or office network, your router or your company's server typically announces "use me (or this address) for DNS" as part of handing out addresses — we'll meet that mechanism (DHCP) in a moment. Behind that first server sits the ISP's or company's resolver, and behind that the public hierarchy. You can see your configured DNS server any time with `ipconfig /all`.

The key insight in that picture is the caching at *every* layer. Your PC caches answers, the nearby DNS server caches answers, and each cached answer carries a **TTL** ("time to live") — a duration, chosen by whoever published the record, after which the answer must be thrown away and fetched fresh (TTL is part of every record in RFC 1035). Caching is why the system is fast. It is also the answer to a question every developer eventually asks:

> **I changed the DNS entry — why is my app still connecting to the old address?**
> Because somewhere between you and the authoritative server, a cache is still serving the old answer and is entitled to do so until its TTL runs out. Nothing is broken; the phone book is just designed to tolerate slightly stale pages in exchange for speed. Locally you can clear *your* machine's cache with `ipconfig /flushdns` — but you can't flush the caches in between. When someone plans a server move and "lowers the TTL a day before," this is exactly what they're managing.

Why localhost never asks anyone

One name from Part 1 deserves a special mention: `localhost`. It resolves to the loopback address (`127.0.0.1`, or `::1` in IPv6) without any DNS query at all — the name is formally reserved in RFC 6761, which says resolvers should answer it with loopback themselves and never send it to the network. Windows makes this visible in a charming way: the default hosts file *comments out* the localhost entries with the note "`localhost name resolution is handled within DNS itself`" — meaning the resolver handles it internally, no file entry and no server needed (Microsoft's default hosts file).

Practical consequence: `localhost` works on a machine with no network cable, no DNS server, no configuration at all. That's why every local development tutorial uses it — it is the one name that cannot be broken by the topics in this post.

DHCP: who gave my machine its address in the first place?

So far we've resolved *other* machines' names. Flip the question around: your own PC has an IP address — where did it come from? You almost certainly never typed it in. It came from DHCP, the Dynamic Host Configuration Protocol (RFC 2131), and the right mental model is a **hotel front desk**.

When your machine joins a network — boots up, wakes from sleep, connects to Wi-Fi — it effectively walks up to the front desk and says "I'd like a room, please." The DHCP server (usually running on your router at home, or a dedicated server in a company) picks a free address from its pool and hands it over — along with the network's other essentials, including *which DNS server to use*. That's the loose end from the DNS section, tied: the front desk doesn't just give you a room number, it also hands you the phone book's address.

The four arrows are the actual DHCP conversation (Discover, Offer, Request, Acknowledge — all defined in RFC 2131), but the dashed box at the bottom is the part that bites developers: the address is a **lease** with an expiry time. A running machine normally renews its lease quietly and keeps the same address. But leave a machine off over a long weekend, or let the lease lapse while the pool is busy, and it can come back with a *different* address — completely by design.

Now connect that to a bug you may have already shipped: you hardcode a colleague's IP — `192.168.1.42` — into a config file because "that's Anna's machine where the test database runs." It works for weeks. Then one morning nothing connects, nobody changed anything, and you burn half a day — because Anna's machine renewed into a different room. **The fix is the whole thesis of this post: refer to machines by name, and let resolution find the current number.**

For machines that *serve* — the database server, the build machine, the license server — networks solve this the other way around: give them a **static address** (configured by hand, outside the pool) or a **DHCP reservation** (the front desk keeps the same room permanently assigned to that guest). Either way, servers get stable addresses, clients get leased ones, and names paper over the difference for everyone.

The toolbox: watch resolution happen

Enough theory — Windows ships everything you need to watch this pipeline work, and it takes five minutes. All commands run in a regular command prompt or PowerShell; only step 5 modifies state.

Step 1 — See your own lease and DNS server

Run `ipconfig /all` and find your active adapter. You'll see your IP address, whether "DHCP Enabled" is Yes, the **DHCP Server** that leased it to you, "Lease Obtained"/"Lease Expires" timestamps, and the **DNS Servers** your resolver asks. This one screen is the whole first half of this post, live.

```
ipconfig /all
```

Step 2 — Ask DNS a question directly

`nslookup` sends a DNS query and shows you the raw answer — which server replied and what addresses came back:

```
nslookup www.embarcadero.com
```

Note what this tool *doesn't* do: `nslookup` talks straight to the DNS server, bypassing your hosts file and local cache. That makes it a scalpel: if `nslookup` returns the right address but your app connects to the wrong one, the problem is local (hosts file or cache), not the DNS server.

Step 3 — Inspect the local cache

Run `ipconfig /displaydns` to dump every answer your resolver has cached, each with its remaining TTL counting down. Visit a website first, run it again, and watch the new entry appear.

```
ipconfig /displaydns
```

Step 4 — Prove the hosts file wins

Add a line to `C:\Windows\System32\drivers\etc\hosts` (editor started as administrator), e.g. `127.0.0.1 testbox`, then `ping testbox`. It resolves instantly to loopback — no DNS server was consulted. Remove the line when you're done.

Step 5 — Flush the cache

When you've changed a hosts entry or a DNS record and want your machine to forget stale answers, clear the cache (run the prompt as administrator):

```
ipconfig /flushdns
```

> **Cross-platform and open-source equivalents**
> Everything above has free counterparts everywhere: on Windows, PowerShell's `Resolve-DnsName` is a more modern `nslookup`; on Linux and macOS the open-source `dig` utility (part of ISC's BIND, the classic open-source DNS server) is the standard tool, and the hosts file lives at `/etc/hosts`. The concepts in this post are identical on every platform — only paths and tool names change.

What this means for your Delphi code

Here's the payoff for a Delphi developer: **your code never resolves names itself, and that's a feature.** When FireDAC connects with `Server=dbserver01` in its parameters, or when you fire a request like this —

```
uses
  System.Net.HttpClient;

var
  LClient: THTTPClient;
begin
  LClient := THTTPClient.Create;
  try
    // "api.mycompany.test" goes through the exact pipeline above:
    // cache -> hosts file -> DNS -> connect to the resulting IP.
    var LResp := LClient.Get('http://api.mycompany.test:8080/status');
    ShowMessage(LResp.ContentAsString);
  finally
    LClient.Free;
  end;
end;
```

— the `THTTPClient` from the modern RTL (its component wrapper is `TNetHTTPClient`) simply hands the hostname to the OS resolver. So does FireDAC, so does every REST client, so does the current Delphi 13 Florence RTL across the board. There is no "Delphi DNS" — there is only the machine's resolution pipeline, which means the classic support calls decode mechanically.

The takeaway from that diagram: both "mysteries" stop being mysteries the moment you remember that resolution happens **per machine, before the connection**. "Works with the IP" means everything from Part 1 is healthy and only the name'number step is sick. "Works over there but not here" means two machines are getting different answers to the same question — and `nslookup` on both, plus a glance at both hosts files, finds the difference nearly every time. On the server itself, `localhost` or the machine's own name resolves trivially; from a client, the full pipeline must cooperate. That asymmetry alone explains a remarkable share of deployment-day surprises.

> **This series: Networking for Delphi Developers**
> 1. IP addresses, localhost, and ports 2. **Names: DNS, the hosts file, and DHCP** — you are here 3. TCP connections and sockets 4. How a web server works: HTTP and WebBroker 5. Web services: REST, JSON, and TMS XData

Takeaways

Names are how humans and configuration files refer to machines; resolution is the pipeline that turns them into the numbers from Part 1 — and now you own that pipeline end to end.

- Your app asks the **OS resolver**; the resolver checks its **cache**, then the **hosts file** (`C:\Windows\System32\drivers\etc\hosts`), then a **DNS server**. First hit wins, and local answers override the network.
- **DNS** is a distributed, cached phone book; an **A record** maps a name to an IPv4 address, and every cached answer expires on its **TTL** — which is why DNS changes take time to "arrive."
- **localhost** is special: reserved to mean loopback, resolved without any server, unbreakable.
- **DHCP** leases your machine its address (and tells it which DNS server to use); leases expire, which is exactly why hardcoded client IPs rot overnight — and why servers get static or reserved addresses while everything else goes by name.
- `ipconfig /all`, `nslookup`, `ipconfig /displaydns`, and `ipconfig /flushdns` let you watch and reset every stage of this from a plain command prompt.

> If Part 1's rule was "address plus port finds the program," Part 2's rule is: **always speak in names, and know the pipeline that turns them into numbers — because that pipeline is where the "mysterious" failures live.**

At this point we can find the machine and the door. In Part 3 we open it: what a TCP connection actually *is*, what a socket is, and what really happens between "connecting…" and "connected."

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —