# DNS Course

Flavio Copes

Updated August 3, 2026

## Contents

Learn how domain names resolve, manage common DNS records, understand delegation and caching, debug failures, and change providers safely.

**What you'll learn:** Configure and inspect a DNS zone, connect a domain to web and email services, trace a lookup, and diagnose stale, missing, or incorrect answers.

## How DNS works

Names, labels, resolvers, root servers, TLDs, authoritative servers, and caches.

### What DNS solves

See DNS as the distributed system that lets a stable domain name point to services whose addresses can change.

You rarely open a website by typing its IP address. You use a name such as `flaviocopes.com`.

The **Domain Name System**, or DNS, is a distributed directory. It lets you ask a precise question such as "what IPv4 addresses belong to this hostname?" or "which servers accept email for this domain?"

The record type is part of the question. These are separate lookups:

```
dig A flaviocopes.com
dig AAAA flaviocopes.com
dig MX flaviocopes.com
```

`A` asks for IPv4 addresses, `AAAA` asks for IPv6 addresses, and `MX` asks where email should be delivered. A successful answer to one question says nothing about the others.

Your device normally sends the question to a **recursive resolver** operated by your network, company, or a public DNS provider. The resolver finds the authoritative source and caches the answer for a limited time. This division makes DNS scale: no single server contains every record and authoritative servers do not handle every browser lookup directly.

A stable name separates an application's public identity from its current infrastructure. You can move a service to a new address and update DNS instead of teaching every client a new name.

DNS only supplies naming information. An address answer does not open a TCP connection, negotiate TLS, or return a web page. Those steps happen afterward, and each can fail while DNS is healthy.

Your action is to query one domain for `A`, `AAAA`, and `MX`. Write down which questions return answers and explain why an empty `AAAA` answer does not prove the domain is broken.

### Names, labels, and fully qualified names

Read a domain name from right to left and recognize the root, top-level domain, registered domain, and subdomain labels.

A DNS name is made of **labels** separated by dots. Read `api.shop.example.com` from right to left.

Starting at the right, `com` is below the DNS root, `example` is below `com`, then come `shop` and `api`. The dots separate labels; they are not part of a label.

The root is represented by the final dot in the complete form:

`api.shop.example.com.`

That absolute name is a **fully qualified domain name**, or FQDN. It identifies the same DNS name regardless of a machine's local search suffix.

A name without the final dot can be treated as relative in some tools and configuration files. On a company network, `printer` might be expanded to `printer.office.example.com`. This convenience can also make debugging confusing because the displayed input is not necessarily the question sent to DNS.

Zone editors add another kind of relativity. Inside the `example.com` zone, a dashboard may expect `api.shop` and append the zone automatically. Another interface may accept the full name. Check its preview so you do not accidentally create `api.shop.example.com.example.com`.

DNS names are case-insensitive for lookup purposes, but the application reached through the name can have case-sensitive URLs or data. Also avoid assuming that every label below a suffix is independently registrable; registration policy and public suffix rules are separate from DNS hierarchy.

Compare absolute and displayed forms:

```
dig A api.shop.example.com.
dig A api.shop.example.com
```

They normally ask the same question when no search behavior intervenes. The trailing dot makes the intent explicit.

Your action is to split `cdn.assets.shop.example.com.` into labels, identify the root and likely zone apex, then write the relative owner name you would enter inside an `example.com` zone editor.

### Follow a DNS lookup

Trace the referral path from the root through a top-level domain to the authoritative server for the requested name.

Suppose a resolver needs the address for `www.example.com` and has nothing cached.

Your laptop does not usually perform the complete search. Its stub resolver asks a recursive resolver, which does the work and returns the final answer.

With an empty cache, the recursive resolver follows referrals:

1. A root server points it to nameservers for the `com` top-level domain.
2. A `com` nameserver points it to nameservers authoritative for `example.com`.
3. An `example.com` nameserver returns the record for `www.example.com`.

The root does not store every website address. It knows where to continue. The `com` server also delegates rather than answering for every `.com` hostname. Each level is responsible for its portion of the namespace.

A referral contains `NS` records. It can also include **glue** address records needed to reach a delegated nameserver whose own name is inside the delegated zone. Glue helps the lookup continue; it is not the final website answer.

If `www.example.com` is a CNAME, the resolver must also resolve the CNAME target. The final authoritative DNS server is not necessarily the web server—authority describes who publishes the DNS data.

You can observe an iterative path with:

```
dig +trace www.example.com A
```

`+trace` makes `dig` follow the chain itself instead of asking your normal recursive resolver to return the completed answer.

Your action is to trace one hostname. Identify the root referral, the top-level-domain referral, the authoritative nameserver, and the final answer or alias.

### Recursive resolvers and caches

Separate the small resolver on your device from the recursive resolver that follows referrals and caches answers for many clients.

Your application normally asks the operating system for a name. The system sends that question to a configured **recursive resolver**.

The small resolver in your device is often called a **stub resolver**. It usually asks for recursion rather than contacting the DNS hierarchy itself. In a `dig` response, the `rd` flag shows that recursion was desired and `ra` shows that the server offers it.

The recursive resolver works on your behalf. On a cold lookup, it can follow referrals from root to top-level-domain to authoritative servers, follow aliases, validate DNSSEC when configured, and return a completed answer.

It also caches the result. The cache key includes at least the name, record type, and class: a cached A answer does not answer an MX question. Each record's TTL limits how long it may be reused.

Watch a cached TTL decrease:

```
dig @1.1.1.1 A flaviocopes.com +noall +answer
dig @1.1.1.1 A flaviocopes.com +noall +answer
```

The second query may be faster and show a smaller remaining TTL. It might also reach a different anycast resolver instance, so identical or higher values do not prove caching is absent.

Negative answers are cached too. A recent NXDOMAIN can remain after you create the missing record. Your browser, operating system, local network, and recursive resolver can each add a cache, which is why “clear the browser” is not a complete DNS diagnosis.

Different resolvers can disagree temporarily because they cached answers at different times or apply different validation and filtering policies. Ask a named resolver explicitly when comparing results.

Your action is to query one A record twice through the same resolver. Record the TTL, query time, `rd`, and `ra` flags, then explain which observations suggest a cache hit and which remain inconclusive.

### Check your understanding: how DNS works

Check names, labels, the root, TLDs, recursive resolvers, authoritative servers, referrals, and caching.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Zones and delegation

Registries, registrars, DNS hosts, zones, nameservers, delegation, and authority.

### Registry, registrar, and DNS host

Distinguish the organizations that operate a TLD, register your domain, and answer DNS queries for your zone.

A **registry** operates a top-level domain such as `.com`. A **registrar** sells registrations and records which nameservers are responsible for your domain.

A **DNS host** runs the authoritative nameservers and gives you a place to manage records. The registrar and DNS host can be the same company, but they do different jobs.

A **web host** is another role: it serves the application after DNS sends clients to it. One company can perform several roles, but keep the controls separate in your mental model.

Use the failing layer to choose the change:

- Moving a website usually means changing A, AAAA, or CNAME records at the DNS host.
- Moving DNS hosting means copying the zone, then changing delegated nameservers through the registrar.
- Moving registration means transferring the domain between registrars; it does not inherently change the website or zone contents.
- Renewing the registration keeps the right to use the domain. Paying a DNS or web-hosting invoice does not renew it.

You can observe the delegation without knowing which dashboard produced it:

```
dig NS example.com +short
dig +trace example.com NS
```

Registration data can be inspected through the registry's RDAP service, while DNS queries show the technical delegation. Privacy-protected registration data may hide personal contact details, but the registration status and nameservers remain operational evidence.

The registrar account is a high-value security boundary because an attacker who changes delegation can redirect many services at once. Use strong authentication, registrar lock features, renewal alerts, and documented recovery access.

Your action is to map one domain to four organizations: registry, registrar, DNS host, and web host. For a future website move, write which dashboard changes and which three should remain unchanged.

### Zones, the apex, and subdomains

Understand a zone as an administrative boundary and know what a dashboard means by the apex or at-sign.

A **zone** is the part of the DNS namespace managed together by one authority. A zone for `example.com` can contain records for the apex and names below it.

The **zone apex** is `example.com` itself. DNS dashboards often represent it with `@`.

A name below the apex is not automatically a separate zone. `www.example.com` and `api.example.com` are usually ordinary records inside the `example.com` zone.

A zone becomes an administrative boundary. Its authoritative servers publish the records in that zone, beginning with required `SOA` and `NS` data. The domain name hierarchy and the zone boundaries can therefore differ.

For example, a company can delegate `team.example.com` to another DNS provider by publishing `NS` records for that child name. After delegation:

- the parent `example.com` zone says which servers know about `team.example.com`
- the child `team.example.com` zone publishes its own records
- `api.example.com` can still remain in the parent zone

Dashboard record names are often relative to the current zone. Entering `www` in the `example.com` zone creates `www.example.com`. Entering `@` creates data at `example.com`. Check the provider preview before saving because some interfaces expect a complete name instead.

Use DNS evidence to see whether a child is delegated:

```
dig NS team.example.com
dig +trace team.example.com
```

An `NS` answer obtained from a recursive cache is useful, while a trace shows where the parent sends the lookup.

Your action is to draw one zone containing an apex, two ordinary subdomains, and one delegated child zone. Mark which provider must be changed for each record.

### Nameservers and NS records

Use NS records to identify the servers authoritative for a zone and understand why changing them moves DNS control.

An **NS record** names a server that is authoritative for a zone.

The parent zone publishes the delegation. For `example.com`, the `.com` zone tells resolvers which nameservers should answer for `example.com`.

The child zone also publishes an NS set at its apex. In a healthy setup, the parent delegation and the child's authoritative data agree. A mismatch can make diagnosis confusing because different queries may expose different sets.

An authoritative nameserver is a DNS service. It is not necessarily the machine hosting the website or receiving email.

When a nameserver's hostname is inside the delegated zone, the parent may need **glue** address records. Without glue, resolving `ns1.example.com` could require first asking the very nameserver whose address is still unknown.

Changing nameservers moves authority, so prepare the destination first:

1. Recreate and verify the complete zone at the new DNS host.
2. Copy the exact assigned nameserver names into the registrar's delegation settings.
3. Keep the old zone available while parent and resolver caches expire.
4. Query the new authoritative servers directly before declaring the move complete.

Do not confuse registrar settings with records in the zone editor. The registrar updates the parent delegation. Adding an NS record only inside the old zone does not normally move the registered domain to a new provider.

Inspect the public result:

```
dig NS example.com +short
dig +trace example.com NS
```

Then save one listed server and query it directly:

```
authoritative_server=$(dig NS flaviocopes.com +short | head -n 1)
dig @"$authoritative_server" flaviocopes.com SOA +norecurse
```

Your action is to record the nameservers returned by an ordinary query and a trace. If they differ, identify whether you are seeing a cache, the parent delegation, or child-zone data.

### SOA records and authority

Read the Start of Authority record as zone metadata rather than treating it as the address of a service.

Every ordinary DNS zone has an **SOA**, or Start of Authority, record.

It describes zone administration. A typical answer contains:

- the primary nameserver field, traditionally called `MNAME`
- a responsible-party value, with the first dot standing in for `@`
- a serial number identifying a zone version
- refresh, retry, and expiry timers used by secondary authoritative servers
- a final value involved in negative-answer caching

The primary field is DNS metadata. It does not identify the website's origin server, and on a managed platform it may not describe the provider's internal source of truth.

The serial should change when zone contents change so secondary servers can detect a newer version. Many managed DNS providers update it automatically. When several authoritative servers give inconsistent answers, compare their SOA serials as one clue:

```
dig +noall +answer SOA flaviocopes.com
authoritative_server=$(dig NS flaviocopes.com +short | head -n 1)
dig @"$authoritative_server" +noall +answer SOA flaviocopes.com
```

The SOA also appears in the authority section of authoritative NXDOMAIN and no-data responses. Resolvers use the SOA TTL and its final field to determine how long the negative result may be cached. This is one reason a recently created name can remain missing for some users.

Do not edit SOA timers casually. Secondary transfer behavior and negative caching affect availability, and managed providers often intentionally control these values.

Your action is to query the SOA through your normal resolver and directly from two authoritative servers. Identify the primary field, serial, and negative-cache value, then compare the serials.

### Check your understanding: zones and delegation

Check registries, registrars, DNS hosts, zones, the apex, subdomains, NS delegation, authority, and SOA records.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## DNS record types

A, AAAA, CNAME, MX, TXT, CAA, SRV, PTR, and the rules that connect them.

### A and AAAA records

Map a hostname directly to an IPv4 or IPv6 address and know when both record types should exist.

An **A record** maps a name to an IPv4 address. An **AAAA record** maps a name to an IPv6 address.

For example:

```
www.example.com.  300  IN  A     192.0.2.40
www.example.com.  300  IN  AAAA  2001:db8::40
```

The owner name is `www.example.com`, `300` is the TTL, and the values are addresses. A record cannot contain `https://`, a port, or a URL path. DNS gets the client to a host; the application protocol handles the rest.

A name can have several A or AAAA records. DNS can return all of them, but that alone is not a health check. Unless another service monitors and removes failed addresses, clients may still receive an unreachable destination.

Modern clients can request both record types and choose a working route based on their network. Publish an AAAA record only when the service, firewall, TLS configuration, and return path really work over IPv6. A broken IPv6 path can make a healthy IPv4 website appear unreliable.

Check the two families separately:

```
dig A www.example.com +short
dig AAAA www.example.com +short
curl -4 https://www.example.com/
curl -6 https://www.example.com/
```

The `curl` checks test more than DNS: they also require routing, a listening service, and valid HTTPS. This separation helps explain why a correct DNS answer does not guarantee a working page.

Your action is to query both address types for a hostname you use. If both exist, test both connection families and record whether the DNS and HTTP evidence agree.

### CNAME records and aliases

Point one name to another name while respecting the rule that a CNAME cannot share its owner name with other data.

A **CNAME record** says one name is an alias of another name.

For example:

```
www.example.com.  300  IN  CNAME  sites.hosting-company.test.
```

The resolver learns that `www.example.com` is an alias, then resolves the target name to the record type the client requested. The target is a hostname, not an IP address.

A CNAME is not an HTTP redirect. The browser still requests `www.example.com`, keeps that name in the URL, and sends it during TLS and HTTP. The hosting service must therefore recognize the original hostname and have a valid certificate for it.

A CNAME cannot coexist with other records at the same name. This is why the zone apex needs special provider features such as flattening when a platform asks for a CNAME-like setup.

The apex already needs `SOA` and `NS` records, so a standard CNAME cannot own `example.com`. Provider features called ALIAS, ANAME, or CNAME flattening synthesize address answers while preserving the required apex data. They are provider behavior, not ordinary CNAME records.

Avoid long CNAME chains. Every additional target can require more lookups, creates another failure dependency, and can make caching harder to reason about.

Inspect both the alias and its destination:

```
dig CNAME www.example.com
dig A sites.hosting-company.test
dig AAAA sites.hosting-company.test
```

`dig +short` may follow and print several values, so use the full answer when you need to distinguish the CNAME from the final addresses.

Your action is to inspect one CNAME used by a hosted service. Identify the alias owner, target, final addresses, and the service that must accept the original hostname.

### MX records and priority

Route email to mail servers and interpret the preference number correctly.

An **MX record** tells senders which mail servers accept email for a domain.

Each MX value contains a preference number and a hostname:

```
example.com.  3600  IN  MX  10 mail1.example.net.
example.com.  3600  IN  MX  20 mail2.example.net.
```

Lower numbers are tried first. Here `mail2` is a fallback when `mail1` cannot be reached; priority `20` does not mean it receives 20 percent of the mail. Equal preferences let senders choose among equally preferred targets.

The target must be a hostname that resolves directly to usable A or AAAA records. Do not put an IP address in the MX value, and do not hide the target behind an HTTP proxy. Mail delivery uses SMTP rather than the web proxy path.

A fallback server is useful only if it is configured to accept and safely relay mail for the domain. An abandoned or weaker fallback can become a security and delivery problem.

Trace the dependency instead of stopping at the MX answer:

```
dig MX example.com +short
dig A mail1.example.net +short
dig AAAA mail1.example.net +short
```

An MX query can succeed while delivery fails because the target has no address, port 25 is unreachable, or the server rejects the recipient. DNS evidence narrows the problem but does not test the complete SMTP exchange.

If a domain intentionally accepts no email, the standardized null MX form is preference `0` with target `.`. Do not invent a fake mail hostname for that purpose.

Your action is to inspect one domain's MX set. Sort the targets by preference, resolve each target's

addresses, and explain which failure would cause a sender to try the next server.

### TXT, CAA, SRV, and PTR records

Recognize four record types used for verification, certificate policy, service discovery, and reverse DNS.

**TXT records** store text used by systems such as domain verification and email authentication. The application using the record defines the text format.

DNS does not interpret a TXT payload. SPF, DMARC, DKIM, and verification systems each define their own owner name and syntax. Long TXT data can be split into several quoted character strings inside one record; consumers concatenate those strings. That is different from publishing several competing policy records.

Query the exact owner, not only the apex:

```
dig TXT example.com
dig TXT _dmarc.example.com
```

**CAA records** say which certificate authorities may issue certificates for a domain. `issue` covers ordinary certificates, `issuewild` covers wildcard policy, and `iodef` can provide an incident-reporting contact. CAA reduces accidental or unauthorized issuance by compliant CAs; it does not revoke existing certificates or replace certificate monitoring.

```
dig CAA example.com
```

**SRV records** advertise a service location with priority, weight, port, and target. The owner includes underscored service and protocol labels, such as `_sip._tcp.example.com`. Lower priority is preferred; weight distributes selection among records with the same priority. SRV helps only when the client protocol knows to query it.

```
dig SRV _sip._tcp.example.com
```

**PTR records** perform reverse lookup from an address to a name under `in-addr.arpa` for IPv4 or `ip6.arpa` for IPv6. The organization controlling the IP range controls this zone, so you normally configure PTR through the server or network provider rather than your ordinary forward-DNS host.

```
dig -x 192.0.2.40
```

Mail systems often expect the PTR name to resolve forward to the same address. A PTR alone does not prove ownership or trustworthiness.

Your action is to inspect one TXT policy, one CAA set, one SRV set, and one PTR. For each, identify who controls the record, which application consumes it, and one thing the record does not guarantee. Use the DNS records explainer when choosing a type.

### Check your understanding: record types

Check A, AAAA, CNAME, MX, TXT, CAA, SRV, PTR, priorities, aliases, and record-owner rules.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Caching and queries

TTL, negative answers, dig, traces, and direct authoritative queries.

### TTL and cached answers

Use Time to Live to control how long recursive resolvers may reuse a record before asking again.

Every DNS record has a **Time to Live**, or TTL. It tells a resolver how many seconds it may cache that answer.

A resolver can reuse the cached answer until that time expires. A full `dig` answer displays the **remaining** TTL, not necessarily the original value configured at authority:

```
dig @1.1.1.1 A flaviocopes.com +noall +answer
```

Run it again and the number often decreases. When it reaches zero, the resolver must refresh before answering again, although serving stale data in failure conditions is a separate resolver feature.

A short TTL makes planned address changes converge sooner and limits how long mistakes remain cached. It also produces more queries and gives caches less ability to absorb authoritative outages. A long TTL reduces query load and can improve resilience, but extends the transition window.

Lower the TTL before a planned migration. Wait for the previous, longer TTL to expire before assuming every cache has adopted the shorter value.

For example, changing a TTL from 86,400 seconds to 300 seconds five minutes before a move does not help a resolver that cached the old answer an hour ago. It may legally reuse that answer for almost another day.

Aliases have multiple cacheable steps. A CNAME and the target's A or AAAA records can have different TTLs. Negative answers have their own caching time derived from the zone's SOA. Diagnose the exact name and record type rather than referring to one universal “domain TTL.”

Compare cache and authority:

```
dig @1.1.1.1 A flaviocopes.com +noall +answer
authoritative_server=$(dig NS flaviocopes.com +short | head -n 1)
dig @"$authoritative_server" A flaviocopes.com +norecurse +noall +answer
```

Authority shows the current published value and its original TTL. A recursive resolver may still show an older value with a countdown.

Your action is to record the same answer and TTL from authority and two recursive resolvers three times over several minutes. Explain every difference without using the phrase “DNS propagation.”

### NXDOMAIN, no data, and negative caching

Distinguish a name that does not exist from a name that exists without the requested record type.

**NXDOMAIN** means the queried name does not exist. A no-data answer means the name exists, but it has no record of the requested type.

The difference appears in a full response:

```
dig A missing.example.com
dig AAAA existing.example.com
```

An NXDOMAIN response has `status: NXDOMAIN`. A no-data response normally has `status: NOERROR` with an empty answer section. The second name might still have an A, MX, or TXT record; it simply lacks the requested AAAA data.

NXDOMAIN applies to the queried name, while no-data is specific to the name and record type. This matters when deciding which additional query can prove the name exists.

Resolvers cache negative answers too. The authoritative response includes the zone's SOA in the authority section. Its TTL and the SOA's negative-cache field determine how long the negative result may remain cached.

Creating a record immediately after someone queried the missing name therefore may not produce an immediate result through that resolver. Lowering the new record's TTL cannot evict an older cached NXDOMAIN response.

`SERVFAIL`, `REFUSED`, and a timeout are different failures. They can indicate DNSSEC validation, authority, policy, or network problems. Calling all of them "DNS propagation" hides the useful evidence.

Compare a recursive answer with authority when a newly created name remains missing:

```
dig @1.1.1.1 A new.example.com
authoritative_server=$(dig NS example.com +short | head -n 1)
dig @"$authoritative_server" A new.example.com +norecurse
```

If authority returns the record and the resolver returns NXDOMAIN with a decreasing negative TTL, wait for that cached result or use a resolver without it. If authority also returns NXDOMAIN, correct the zone.

Your action is to find or create one no-data example and one nonexistent name. Record the status, answer count, SOA in the authority section, and the negative TTL evidence for each.

### Query DNS with dig

Ask for one record type, use a chosen resolver, and read the answer, status, flags, and remaining TTL.

The `dig` command gives you a direct view of a DNS response.

Ask for an address record:

```
dig A flaviocopes.com
```

Read the complete response before shortening it. The header contains evidence such as:

- `status`: `NOERROR`, `NXDOMAIN`, `SERVFAIL`, or another result code
- flags: `aa` means the answer is authoritative, while `ra` says recursion is available
- `ANSWER`: records that answer the question
- `AUTHORITY`: delegation or authority information, including the SOA for negative answers
- `SERVER`: the resolver that replied
- `Query time`: how long this request took

The number beside a returned record is its remaining TTL in seconds. A recursive resolver decrements this value while the answer is cached.

`NOERROR` does not guarantee that the answer section contains the requested type. The name may exist without an A record.

Use `+short` only when you need values for a script or quick check. It hides the status, flags, authority section, and server:

```
dig A flaviocopes.com +short
```

Add a resolver after `@` when you want to compare caches:

```
dig @1.1.1.1 A flaviocopes.com +short
dig @8.8.8.8 A flaviocopes.com +short
```

For focused evidence without losing record TTLs, select sections explicitly:

```
dig A flaviocopes.com +noall +answer +comments
```

If two resolvers disagree after a change, compare their remaining TTLs and then query an authoritative server. Repeatedly refreshing your browser tells you much less.

Your action is to run one full `dig` query and annotate the status, flags, answer, TTL, responding server, and query time. Then run `+short` and list the evidence it removed.

### Trace resolution and query authority

Use dig trace and direct nameserver queries to separate delegation problems from recursive-cache problems.

Run a trace to follow referrals from the root:

```
dig +trace flaviocopes.com
```

This makes `dig` perform iterative queries itself. It is different from asking your configured recursive resolver, so it is useful for seeing the delegation path. It does not show what a particular user's resolver still has cached.

Then find the authoritative nameservers and ask one directly:

```
dig NS flaviocopes.com +short
authoritative_server=$(dig NS flaviocopes.com +short | head -n 1)
dig @"$authoritative_server" A flaviocopes.com +norecurse
```

In the direct response, look for the `aa` flag and inspect the answer rather than assuming that any responding server is authoritative.

These three views isolate different layers:

1. `dig A flaviocopes.com` shows what your configured recursive resolver returns.
2. `dig +trace flaviocopes.com A` follows root and parent referrals.
3. `dig @"$authoritative_server" A flaviocopes.com +norecurse` asks the published source directly.

If authority has the new answer but a recursive resolver has the old one, the resolver probably has a cached answer whose TTL has not expired. If the trace reaches unexpected nameservers, investigate the parent delegation. If the correct authoritative server returns the wrong value, fix the zone itself.

Query at least two authoritative servers when results are inconsistent. One secondary may have stale data or a transfer problem. Also distinguish a valid negative answer from `SERVFAIL` or a timeout; they point to different causes.

`+trace` can fail on networks that block the direct DNS traffic it needs even when normal recursive lookups work. Treat every command as one piece of evidence.

Your action is to collect the recursive, trace, and direct-authority views for one hostname. Write a one-line diagnosis for each possible disagreement between them.

### Check your understanding: caching and queries

Check TTL, planned changes, negative caching, NXDOMAIN, no-data answers, dig, traces, and authoritative queries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Websites and safe changes

Connect websites, understand DNS proxies, migrate records, and debug failures.

### Connect a domain to a website

Turn hosting-provider instructions into the smallest correct set of DNS records and verify the result layer by layer.

Your hosting provider should tell you which record names and values to add. Common setups use an A record for an IP address or a CNAME for a provider hostname.

Translate each instruction literally:

- an A value must be an IPv4 address
- an AAAA value must be an IPv6 address
- a CNAME value must be a hostname
- `@` usually means the zone apex
- `www` normally means `www.example.com` inside the `example.com` zone

Do not paste a URL such as `https://host.example/path` into an address or CNAME field. DNS records contain names and addresses, not schemes, ports, or paths.

Decide whether both the apex and `www` should work. They are different DNS names and may need separate provider configuration. An HTTP redirect from one to the other happens after DNS; a CNAME does not change the browser URL.

Before saving, inspect records already using the owner name. A standard CNAME cannot coexist with A, AAAA, TXT, or MX data at the same name. Remove an old record only after identifying the service that depends on it. A forgotten verification or mail record can matter even when it does not serve the website.

Verify one layer at a time:

```
dig A example.com
dig AAAA example.com
dig CNAME www.example.com
curl -I https://example.com/
```

First confirm authority publishes the expected record, then compare recursive resolvers. After DNS is correct, test the TCP connection, TLS certificate, and HTTP response. A correct address does not prove the web server recognizes the hostname.

If a proxy is enabled, public DNS should return proxy addresses rather than the origin. In that case, an HTTP proxy error can mean the origin, certificate mode, or application is wrong while DNS is behaving exactly as configured.

Your action is to turn one hosting provider's domain instructions into a table with owner, type, value, proxy status, and purpose. Verify the final DNS answer and HTTPS response separately.

### DNS-only and proxied records

Understand when a DNS provider returns your origin address and when it returns proxy addresses that receive web traffic first.

A DNS-only record returns the configured destination. The client then connects to that address.

With a proxied web record, Cloudflare returns Cloudflare anycast addresses instead of the configured origin address. The request path becomes:

```
browser -> Cloudflare proxy -> origin server
```

The proxy can apply caching, DDoS protection, WAF rules, redirects, and edge TLS behavior. This also means a `dig` result for a proxied hostname should not match the origin IP.

With DNS-only mode, the path is direct:

```
client -> configured destination
```

The origin address is publicly visible, and Cloudflare cannot apply its HTTP protections or caching to that traffic.

Only eligible A, AAAA, and CNAME records serving supported HTTP or HTTPS traffic can use the ordinary proxy. MX and TXT records are always DNS-only. The A or AAAA hostname used by a mail server must also remain DNS-only so SMTP clients reach the mail server rather than Cloudflare's web proxy.

Proxying adds another contract. Cloudflare must be able to reach the origin, the origin must accept the requested hostname, and its TLS mode and certificate must match the configuration. A healthy DNS answer can still lead to a proxy error when that origin connection fails.

Do not rely only on hiding the origin address. Restrict origin access to intended proxy traffic where practical, and ensure no DNS-only sibling record exposes the same origin unintentionally.

Compare evidence:

```
dig A www.example.com +short
curl -I https://www.example.com/
```

The first command shows where clients connect. The second exercises the proxy and origin path. An HTTP error does not automatically mean DNS is wrong.

Your action is to classify five records—website A, website CNAME, MX, domain-verification TXT, and mail-server A—as proxied or DNS-only, and explain the traffic path for each.

### Change DNS records safely

Prepare a reversible record change, account for the old TTL, verify both endpoints, and keep the previous service available during the transition.

Before a planned move, inventory the current records and lower the relevant TTL. Do this early

enough for the old TTL to expire.

Lowering a TTL does not remove answers already cached with the old, longer TTL. If the current TTL is 24 hours, lower it at least 24 hours before the move. Record the original value so you can restore it later.

Prepare the new service before changing DNS. For a website, verify its application, certificate, redirects, and data independently of public DNS. You can direct one HTTPS request to the new address while retaining the real hostname:

```
curl --resolve www.example.com:443:203.0.113.40 \
  https://www.example.com/
```

This tests the new endpoint with the correct TLS and HTTP hostname. It does not test the future public DNS answer.

Use a written change plan:

1. Export or record the current A, AAAA, CNAME, MX, and important TXT records.
2. Define the expected new answer and a rollback condition.
3. Lower the relevant TTL early and wait out the old TTL.
4. Test the new service.
5. Change the authoritative record.
6. Query authority directly, then compare several recursive resolvers.
7. Keep the old service working until old positive and negative caches have expired.
8. Restore a normal TTL after the result is stable.

Changing nameservers is a larger operation than changing one address record. Parent delegation caches are involved, and DNSSEC-enabled domains must coordinate DS data carefully. A stale DS record pointing at a newly unsigned or differently signed zone can cause validating resolvers to return `SERVFAIL`.

Do not judge completion only from your laptop. Its operating system, browser, router, and resolver may each cache data.

Your action is to write a reversible runbook for moving one test hostname. Include the old TTL deadline, preflight request, authority query, two resolver queries, rollback trigger, and time to retire the old endpoint.

### Debug a DNS failure

Use a fixed sequence to locate mistakes in registration, delegation, authority, caching, proxying, TLS, or the application.

Start with the exact hostname and record type. Check registration and delegated nameservers, then trace the lookup.

Do not begin by changing records. Collect evidence from the top down:

1. **Name:** copy the complete failing hostname. `example.com` and `www.example.com` are separate.
2. **Question:** identify the required A, AAAA, CNAME, MX, or TXT record type.
3. **Delegation:** use `dig +trace` to see where the parent sends queries.
4. **Authority:** ask a listed authoritative server directly with `+norecurse`.
5. **Cache:** compare the authoritative answer with public and local recursive resolvers.

6. **Connection:** test whether the returned address accepts the required port.
7. **TLS:** inspect certificate name, validity, and proxy TLS mode.
8. **HTTP:** inspect the status, redirect location, and application response.

Useful commands are:

```
dig +trace www.example.com A
authoritative_server=$(dig NS example.com +short | head -n 1)
dig @"$authoritative_server" www.example.com A +norecurse
dig @1.1.1.1 www.example.com A
curl -Iv https://www.example.com/
```

Interpret the response precisely. `NXDOMAIN` means the name is missing. `NOERROR` with no requested records is no-data. `SERVFAIL` can indicate broken authority or DNSSEC validation. A timeout suggests a network or server reachability problem. Different answers with decreasing TTLs suggest caching.

If DNS is correct, stop editing DNS. A certificate mismatch belongs to TLS. A `502` from a proxy means the client reached the proxy but it could not complete the origin request. A redirect loop after enabling a proxy is usually an HTTP or TLS-mode disagreement.

When public DNS is intentionally unchanged, test a candidate web origin with `curl --resolve`. This isolates the new endpoint without poisoning local DNS or waiting for caches.

Your action is to create a diagnostic worksheet for one hostname with one row per layer. Include the command, observed evidence, expected evidence, and next owner for a failure at that layer.

### Check your understanding: websites and changes

Check website records, conflicting data, DNS-only and proxied responses, TTL planning, migrations, and layered debugging.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Email and DNS security

SPF, DKIM, DMARC, DNSSEC, encrypted DNS, and a final zone review.

### SPF records

Authorize the systems allowed to send email for a domain without publishing conflicting policies or exceeding lookup limits.

SPF uses a TXT record to publish which systems may use a domain in the SMTP envelope sender.

That envelope domain can differ from the human-visible `From:` address. DMARC adds alignment rules for the visible domain, so an SPF pass alone does not prove that the displayed sender is authorized.

A policy might contain IP mechanisms and an included provider policy:

```
v=spf1 ip4:192.0.2.0/24 include:_spf.mail-provider.test -all
```

Read it from left to right. The mechanisms describe allowed senders, and `-all` says unmatched senders should fail. Copy the provider's current value and understand its rollout guidance before using a strict ending on a production domain.

Keep exactly one SPF policy per owner name. When adding a sender, merge its mechanism into the existing `v=spf1` record instead of publishing a second one. Multiple selected SPF records produce a permanent error rather than combining their permissions.

SPF limits evaluation to ten DNS-causing mechanisms and modifiers. Nested `include`, `a`, `mx`, `redirect`, and related lookups count toward that budget. A convenient chain of providers can exceed the limit even when the TXT text looks short. Direct `ip4` and `ip6` mechanisms do not require DNS lookups during evaluation.

SPF also has operational limits. Forwarding can change the connecting server and break SPF unless the forwarding system rewrites the envelope sender. SPF does not sign message content; DKIM performs that separate job.

Inspect the published policy:

```
dig TXT example.com +short
```

Look for one string beginning with `v=spf1`, then expand every `include` while counting DNS-causing terms. Remove services you no longer authorize. If a service uses a subdomain as its envelope sender, query that exact subdomain too.

Your action is to audit one domain's SPF policy. List each authorized sender, count the DNS-causing terms including nested includes, confirm there is one policy, and identify the final `all` qualifier.

### DKIM and DMARC

Verify signed mail, require alignment with the visible From domain, and roll out an enforcement policy without blocking legitimate senders.

DKIM signs selected message content. The receiving server gets the public key from a DNS record beneath a selector such as `mailer._domainkey.example.com`.

The sending service keeps the private key and adds a `DKIM-Signature` header. Its `d=` value names the signing domain and `s=` names the selector. The receiver combines them to query the public key:

```
dig TXT mailer._domainkey.example.com +short
```

Selectors let you rotate keys or give separate services separate keys. Publish the new key before sending with it, retain the old key while messages signed with it can still arrive, and remove compromised or retired keys deliberately.

A valid DKIM signature proves that the signed headers and body survived verification under that signing domain. It does not encrypt the message, prove the human sender's identity, or say the content is trustworthy.

DMARC connects authentication to the domain visible in the `From:` header. A message passes DMARC when at least one of these is true:

- SPF passes and its authenticated domain aligns with the visible From domain
- DKIM passes and its `d=` domain aligns with the visible From domain

This alignment is why an unrelated provider domain can pass SPF or DKIM while DMARC still fails.

A monitoring policy looks like:

```
_dmarc.example.com. IN TXT "v=DMARC1; p=none; rua=mailto:dmarc@example.com"
```

`p=none` requests reports without asking receivers to quarantine or reject failures. Review aggregate reports, identify every legitimate sender, fix alignment, then move deliberately toward `quarantine` or `reject`. Forwarders and mailing lists can modify message paths or content, so test real mail flows instead of assuming one authentication method always survives.

Reporting addresses receive operational data and should be secured and monitored. DMARC policies are receiver requests, not a guarantee that every receiver makes the same delivery decision.

Your action is to inspect a delivered message's authentication results and DNS. Identify the SPF domain, DKIM `d=` and `s=`, visible From domain, alignment result, and current DMARC policy. Use the email DNS tool to check the record structure.

### What DNSSEC protects

Understand signed DNS data, the chain of trust, and the important security properties DNSSEC does not provide.

DNSSEC adds signatures that let a validating resolver check the origin and integrity of DNS data.

Signed zones publish `DNSKEY` records and `RRSIG` signatures over record sets. A validating resolver checks that the signature matches the data and a trusted key. This detects forged or modified answers; it does not hide the records.

Trust crosses a delegation through a `DS` record in the parent zone. The parent vouches for a key in the child, and the chain continues toward a resolver's configured root trust anchor.

The resolver can classify a response as:

- **secure:** signatures validate through a trusted chain
- **insecure:** the zone is deliberately unsigned, with proof that no DS delegation exists
- **bogus:** signatures or the chain should validate but do not

A bogus result is normally returned to applications as `SERVFAIL`. This is a safety feature: the resolver refuses data it cannot authenticate.

Inspect signed response data with:

```
dig +dnssec A cloudflare.com
dig +dnssec DNSKEY cloudflare.com
dig DS cloudflare.com
```

`+dnssec` asks to receive DNSSEC records; it does not make `dig` perform complete validation by itself. An `ad` flag says the responding recursive resolver considers the answer authenticated. That signal is meaningful only when you trust the path to that resolver.

DNSSEC does not encrypt queries, hide record contents, protect an unsigned zone, keep an authoritative server online, or make a malicious application safe. It authenticates DNS data, not the service reached afterward.

Deployment order matters. Have the DNS host sign and verify the zone before publishing DS data through the registrar. During removal or provider migration, coordinate keys and DS records so the parent never promises a chain the active child cannot satisfy. A stale DS record can make the entire zone fail for validating users while non-validating tests appear healthy.

Your action is to query one signed and one unsigned domain through a validating resolver. Record the RRSIG, DNSKEY, DS, and `ad` evidence, then explain why an unsigned answer is not automatically bogus.

### Encrypted DNS and a final zone review

Separate DNSSEC validation from encrypted transport, then inspect one real zone from delegation through web, mail, and security records.

Traditional DNS queries are often visible on the network. DNS over TLS and DNS over HTTPS encrypt the connection between a client and a supporting resolver.

DNS over TLS, or DoT, carries DNS through TLS on a dedicated service. DNS over HTTPS, or DoH, carries DNS queries inside HTTPS. Both can prevent a local network observer from reading or modifying that client-to-resolver exchange.

Encryption changes who can observe the query; it does not eliminate trust. The selected resolver can still see the requested names, apply filtering, log metadata, and perform the remaining authoritative lookups. Those upstream lookups are not automatically encrypted just because the first hop used DoH or DoT.

DNSSEC solves a different problem:

- DoH and DoT protect transport between two endpoints.
- DNSSEC lets a validating resolver authenticate signed DNS data.

Encrypted transport without validation can carry an incorrect answer securely. DNSSEC without encrypted transport can authenticate an answer while leaving the query visible. A deployment can use both.

Encrypted DNS can also affect operations. A browser using its own DoH resolver may bypass the operating system resolver, company split-horizon names, or local filtering. During diagnosis, identify the actual resolver instead of assuming every application uses the network default.

For the final review, choose a domain you control or can inspect safely. Record its delegated nameservers, SOA, A and AAAA answers, aliases, MX and TXT records, TTLs, DNSSEC state, and web response.

Build the review from evidence:

```
dig +trace example.com
dig SOA example.com
dig A example.com
dig AAAA example.com
dig MX example.com
dig TXT example.com
dig +dnssec A example.com
curl -I https://example.com/
```

For every answer, name its owner, type, value, TTL, authoritative provider, and consuming service. Separate observations from conclusions: an A answer proves address data exists, not that HTTPS works; an SPF record proves policy is published, not that every legitimate sender passes DMARC.

Finish by proposing one safe change. Include the old and new values, the current TTL, when to lower it, how to test the new service before cutover, direct-authority verification, resolver comparisons,

rollback trigger, and the time the old service can be retired.

Your action is to produce that one-page zone review. End with the three most important risks you found and the exact evidence that would prove each risk has been removed.

### Check your understanding: email and DNS security

Check SPF, DKIM, DMARC, alignment, DNSSEC, chains of trust, encrypted DNS, and the final zone review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

---

The interactive version of this course is available at https://flaviocopes.com/courses/dns/.