Tailscale Course

Flavio Copes

Updated August 3, 2026

Contents

Build and operate a private Tailscale network with identity, access policies, MagicDNS, SSH, subnet routers, exit nodes, automation, and practical troubleshooting.

What you'll learn: Connect a laptop and Linux server, restrict access with grants, publish a private service, add routed resources, automate enrollment, and diagnose the complete path.

Tailscale foundations

Understand tailnets, identity, WireGuard, control and data planes, direct paths, and DERP relays.

Understand the problem Tailscale solves

See why connecting private devices across changing networks is difficult and what Tailscale removes from a traditional VPN setup.

A traditional remote-access VPN usually sends clients through one gateway. You must expose, patch, scale, and route that gateway before two private devices can communicate.

That gateway is a single point of failure and a public attack surface. It also shapes your traffic: two laptops in the same city may talk through a datacenter on another continent. Every new device means more certificates, more routing rules, more firewall changes.

What makes private connectivity hard

Most of your devices sit behind NAT. Your laptop has a private address on your home Wi-Fi. Your server may sit behind a cloud firewall. Neither can accept an inbound connection unless you forward ports or expose services publicly.

The classic workaround looks like this:

```
# expose SSH publicly and hope the firewall rules hold
ssh flavio@203.0.113.40 -p 22022
```

You changed the port, added fail2ban, maybe restricted source IPs. You are still running a public service, and you maintain that protection forever.

What Tailscale changes

Tailscale creates an identity-aware overlay network called a tailnet. Devices keep their existing Internet connections, then use Tailscale addresses and policies for private communication. You do not open an inbound port on every laptop or move all traffic through one central server.

Each device authenticates with your identity provider, gets a stable private address, and builds encrypted WireGuard tunnels directly to the peers it needs. The same SSH connection becomes:

```
ssh flavio@lab-server
# no public port, no port forwarding, no central gateway
```

The server's public firewall can drop every inbound packet, and this still works, because both sides connect outward.

Try it

Draw your laptop, one server, their ordinary Internet paths, and the private path you want between them. Mark which public ports you would otherwise need to expose.

If the answer is "SSH on the server, and maybe a web dashboard", you found exactly what a tailnet removes from the public Internet.

Meet the tailnet

Model a tailnet as a private collection of users, devices, services, routes, names, and policies rather than one tunnel endpoint.

A **tailnet** is the private network Tailscale creates for an account or organization. Authenticating a new device adds it to that network.

Think of it as one flat private network that follows your identity, not a place. Your laptop at home, your phone on mobile data, and a server in a datacenter all sit on the same tailnet.

What every device gets

Each device receives stable Tailscale IPv4 and IPv6 addresses plus a private DNS name. The IPv4 address comes from the `100.64.0.0/10` range, so it never collides with your home or office network.

```
tailscale status
```

```
100.87.12.44   flavio-mbp    flavio@   macOS   -
100.101.9.23   lab-server    flavio@   linux   active; direct 203.0.113.40:41641
100.114.3.7    pixel-9       flavio@   android offline
```

Those addresses do not change when the device moves between networks. That stability is what makes it safe to put them in SSH configs and connection strings.

Users, devices, and services

Users come from an identity provider, while tagged devices represent services. A laptop belongs to a person. A production server should not: you tag it (for example `tag:server`) so its identity describes its role, not whoever happened to install it.

The admin console coordinates membership, DNS, routes, and access policy for the whole tailnet. It is the one place where you see every device, who owns it, and when it was last seen.

What stays outside

A tailnet is private by definition. Visitors to your public website never touch it. Only enrolled devices participate, and later lessons decide what each of them may reach.

List the people, laptops, phones, servers, and private services that belong in a small practice tailnet. Leave public website visitors outside it. That list becomes your enrollment plan for the next lessons: each entry is either a person who logs in or a service that gets a tag.

Separate the control and data planes

Trace what the Tailscale coordination service manages and what remains encrypted between devices in the data plane.

Tailscale separates coordination from packet transport. This distinction explains both how the system works and what you trust Tailscale to manage.

The control plane

The control plane distributes identities, public keys, policies, routes, DNS settings, and endpoint information. It is the coordination service Tailscale runs. When your laptop joins, it tells the control plane ”here is my public key, here are the endpoints where you can reach me”, and receives the same information about every peer it is allowed to see.

The coordination service does not sit in the normal data path or hold device private keys. Private keys are generated on each device and never leave it.

The data plane

The data plane runs on each device and carries WireGuard-encrypted packets. When you SSH from your laptop to a server, the packets flow between the two machines (or through a relay when a direct path fails). They do not pass through Tailscale's servers as readable traffic.

You can watch the data plane at work:

```
tailscale ping lab-server
```

```
pong from lab-server (100.101.9.23) via 203.0.113.40:41641 in 12ms
```

That `via 203.0.113.40:41641` is a direct UDP path between your two devices. The control plane helped them find each other, then stepped aside.

Why the split matters

If the coordination service becomes unreachable, existing connections keep working. Devices already hold each other's keys and endpoints. What breaks is change: new devices cannot join, and policy updates stop propagating until it returns.

The split also defines your trust boundary. Tailscale can see which devices exist and which endpoints they use. It cannot read your traffic, because it never holds the private keys that would decrypt it.

Take a file transfer between two devices and label key distribution, policy distribution, encryption, packet transport, and decryption as control-plane or data-plane work. The first two belong to the control plane. Everything else happens on your own devices.

Understand direct connections and DERP

Learn how NAT traversal attempts a direct UDP path and why an encrypted DERP relay is sometimes the correct fallback.

Tailscale tries to create a direct UDP path between peers. NATs and firewalls do not always allow that path.

How the direct path is built

Both devices learn each other's candidate endpoints from the control plane: LAN addresses, public addresses, and ports discovered through STUN. Then they send packets toward all of them and keep whichever path works. This is NAT traversal, and it succeeds surprisingly often, even when both devices sit behind home routers.

You can check what your current network allows:

```
tailscale netcheck
```

```
Report:
    * UDP: true
    * IPv4: yes, 93.44.120.15:41641
    * MappingVariesByDestIP: false
    * Nearest DERP: Frankfurt
```

`UDP: true` is good news. A network that blocks outbound UDP entirely forces everything through relays.

When DERP takes over

When direct connectivity fails, Tailscale can relay the already encrypted traffic through a DERP server. DERP (Designated Encrypted Relay for Packets) servers run in regions around the world, and connections often start on DERP while the direct path is still being negotiated.

The relay sees connection metadata but cannot decrypt the WireGuard payload. Your packets were encrypted on your device with a key the relay never had.

```
tailscale ping strict-office-box
```

```
pong from strict-office-box (100.99.4.18) via DERP(fra) in 48ms
```

A relayed connection still works, although latency and throughput may be worse than a direct path. Hard NATs, symmetric NATs, and UDP-blocking corporate firewalls are the usual causes.

The honest promise

Write down the honest promise: Tailscale prefers direct peer connections, but it does not guarantee that every connection is direct.

A DERP path is not a failure, and it is not a security downgrade. It is the designed fallback that keeps the connection alive when the network refuses to cooperate. Later in the course you will learn when a relayed path is worth improving and when it is fine to leave alone.

Check your understanding: tailscale foundations

Check tailnets, identity, WireGuard, control and data planes, direct paths, DERP relays, and the limits of the trust model.

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

Take this interactive quiz online

Connect your tailnet

Create a tailnet, add a laptop and Linux server, inspect addresses, and use MagicDNS.

Create a practice tailnet

Create a personal tailnet, inspect its identity boundary, and confirm the current free-plan terms before adding real devices.

Signing in to Tailscale creates your first tailnet. Use an identity you control and a disposable server for this course.

The Personal plan currently permits six users in one tailnet, but plan details can change. Check the official pricing page before designing around a limit. Open the admin console and locate the Machines, Users, DNS, Access controls, and Keys pages.

Record the tailnet name and the identity provider that controls admission. Do not store login tokens or screenshots containing secrets in the project notes.

After signing in on two disposable devices, inspect their state:

```
tailscale status
tailscale ip -4
tailscale netcheck
```

Record each device name and Tailscale address. Confirm which identity enrolled it. Remove any old test device you no longer control. A clean inventory makes later policy tests understandable because every node has one known owner and purpose.

Connect your laptop

Install the Tailscale client on your daily computer, authenticate it, and inspect the private addresses assigned to the device.

Install the official Tailscale client for your operating system, sign in, and connect it to the practice tailnet.

On macOS, use the App Store app or the standalone package from tailscale.com. On Windows, run the installer. On Linux, the install script sets up the package repository for you. After installing, sign in through the app, or run:

```
sudo tailscale up
```

The client creates a virtual network interface and receives stable tailnet addresses. It does not replace your normal Wi-Fi or Ethernet interface. Your regular Internet traffic keeps flowing exactly as before. Only traffic addressed to tailnet devices uses the new interface.

Inspect what the device received

Use the application or CLI to confirm the device is connected before changing routes or access policy.

```
tailscale status
tailscale ip -4
tailscale ip -6
```

```
100.87.12.44   flavio-mbp   flavio@   macOS   -
```

```
100.87.12.44
fd7a:115c:a1e0::4401:c2c
```

The IPv4 address comes from the `100.64.0.0/10` range, and the IPv6 address from a Tailscale-specific prefix. Both stay with this device for as long as it remains in the tailnet, no matter which network it joins.

Match it in the admin console

Open the admin console and find your laptop in the machines list. Match the addresses in the terminal with the device entry in the admin console. Same IP, same name, same user. That confirmation matters more than it seems, because later lessons make policy decisions based on these identities.

Rename the device if its generated name is unclear. `flavios-macbook-pro-3` is a bad name to type into an SSH config; `flavio-mbp` is better. The MagicDNS name follows the machine name, so pick one you are happy to use everywhere.

One check before moving on: if `tailscale status` prints `Logged out` or `Stopped`, run `sudo tailscale up` again and complete the browser login. Nothing in the next lessons works from a stopped client.

Connect a Linux server

Install Tailscale on a disposable Linux server, authenticate it interactively, and verify that no public application port was required.

The second node will be a disposable Ubuntu or Debian server. Keep its existing recovery access until the private path is tested.

That last sentence is the rule I want you to internalize. Never cut off the current way in before the new way is proven. If your only access today is SSH over the public address, keep that session open through this whole lesson.

Install and authenticate

Use Tailscale's official installer or the documented package repository, then run `tailscale up`:

```
curl -fsSL https://tailscale.com/install.sh | sh
sudo tailscale up
```

The command prints a browser authentication URL:

```
To authenticate, visit:

    https://login.tailscale.com/a/1b2c3d4e5f6a7
```

Open it from your laptop, approve the device, and the command returns. After approval, the server appears in the same tailnet without requiring an inbound Tailscale port on the server firewall. The client opened an outbound connection to the coordination server. Nothing new was exposed publicly.

Verify from both sides

On the server:

```
tailscale status
```

```
100.101.9.23   lab-server   flavio@   linux   -
100.87.12.44   flavio-mbp   flavio@   macOS   active; direct 93.44.120.15:41641
```

Confirm the server appears in the admin console too, owned by your user and recently seen. Then, from your laptop, test the private path while the old session stays open:

```
ssh ubuntu@100.101.9.23
```

Keep the original SSH session open while testing the new private route. If the tailnet connection works, you now have two independent paths to the machine. If routing or policy breaks something later, the public session is your recovery path.

One common surprise: a strict host firewall can block the new path. Tailscale traffic arrives on the `tailscale0` interface, so a `ufw` ruleset that only allows the public SSH port will reject SSH over the tailnet IP until you allow it on that interface.

Use MagicDNS and test the path

Reach the server by its MagicDNS name, compare name and address tests, and prove the connection uses the tailnet path.

MagicDNS registers a private name for each tailnet device. New tailnets normally have it enabled by default.

A short hostname such as `lab-server` is easier to use than its Tailscale IP. The full name includes your tailnet domain, something like `lab-server.tail4a5b6.ts.net`, but inside the tailnet the short form resolves too.

Test the name

Start with a plain connection test:

```
ssh ubuntu@lab-server
```

If SSH connects, the name resolved to the Tailscale address and the private path works. Names beat raw addresses in every config file you will write from now on: they survive device replacement, and they make policies and logs readable.

Test the path, not just reachability

Test the name, then use `tailscale ping` to see whether the path is direct or relayed. Ordinary `ping` tests IP reachability but does not explain Tailscale path negotiation.

```
tailscale ping lab-server
```

```
pong from lab-server (100.101.9.23) via 203.0.113.40:41641 in 14ms
```

`via <ip>:<port>` means a direct connection. If you see `via DERP(fra)` instead, traffic is relayed through Tailscale's Frankfurt relay: still encrypted, still working, just not direct. The first ping sometimes goes through DERP while the direct path is negotiated. Run it again and watch it switch.

Prove the old path was not used

Connect by name, record whether Tailscale reports a direct or DERP path, and confirm the server's public SSH address was not used:

```
ssh ubuntu@lab-server 'echo $SSH_CONNECTION'
```

```
100.87.12.44 52441 100.101.9.23 22
```

Both addresses are tailnet addresses. The connection entered through the Tailscale interface, not the public one.

If name resolution fails, check that MagicDNS is enabled in the admin console DNS page and that the client accepts DNS settings. When `tailscale ping` works by IP while the name fails, the problem is DNS, not connectivity. That distinction saves you from debugging the wrong layer.

Check your understanding: connect your tailnet

Check account creation, client installation, device authentication, Tailscale addresses, status inspection, MagicDNS, and first-path verification.

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

Take this interactive quiz online

Control access

Replace broad connectivity with grants, groups, tags, tests, and least-privilege rules.

Move from connectivity to authorization

Separate membership in a tailnet from permission to reach a particular device, protocol, port, or application capability.

Joining the tailnet answers ”who or what is this?” Access policy answers ”what may it reach?” Those are different decisions.

Up to now, everything in your practice tailnet could reach everything. That is because new tailnets ship with a permissive starter policy, so the first experience works without configuration:

```
{
  "grants": [
    {
      "src": ["*"],
      "dst": ["*"],
      "ip":  ["*"]
    }
  ]
}
```

Every device may reach every device on every port. Fine for a two-device experiment. Wrong for anything you plan to keep.

Deny by default

Tailscale access controls are deny by default once you replace the permissive starter policy. Remove that wildcard grant and no packet flows anywhere until a rule allows it.

There is no ”block” rule type. Rules grant access; they do not add explicit deny entries. Anything not granted is denied. This model has one big consequence: you can read a policy and know exactly what is reachable. There is no rule-ordering puzzle where a later deny overrides an earlier allow.

What a grant contains

Modern policies use grants for network and application capabilities. A grant names sources, destinations, and allowed IP protocols or ports:

```
{
  "src": ["flavio@example.com"],
  "dst": ["tag:server"],
  "ip":  ["tcp:22"]
}
```

Sources and destinations can be users, groups, tags, or autogroups. The `ip` field narrows the grant to protocols and ports. We will write and apply a real one in the next lesson.

Start from a sentence

Before touching JSON, write one sentence describing who should reach the lab server, on which port, and why. Everything else should remain ungranted.

Mine reads: ”My user may reach TCP port 22 on servers tagged `tag:server`, to administer them over SSH.” If you cannot write the sentence, you are not ready to write the rule. The sentence forces you to name the source, the destination, and the purpose, which is exactly what a reviewer will ask about later.

Write your first grant

Replace broad access with one grant that permits your user to reach SSH on the tagged lab server and nothing more.

Start with one narrow connection: your user may reach TCP port 22 on the lab server.

One rule, one port, one destination. If this works, you understand the whole model. Everything after is repetition with different values.

Prepare the tag

Use your real tailnet identity as the source and a server tag as the destination. Tags need owners before they can be assigned, so declare that first in the policy file:

```
{
  "tagOwners": {
    "tag:server": ["flavio@example.com"]
  }
}
```

Then tag the lab server, either from the admin console machine list or from the machine itself:

```
sudo tailscale up --advertise-tags=tag:server
```

Once tagged, the node's identity is its role, not your user account.

Write the grant

The `ip` capability names the protocol and port. Add this to the `grants` section:

```
{
  "grants": [
    {
      "src": ["you@example.com"],
      "dst": ["tag:server"],
      "ip": ["tcp:22"]
    }
  ]
}
```

Replace `you@example.com` with your identity. Save from an existing recovery session so a policy mistake does not remove your only management path. If you edit the policy over tailnet SSH and the edit breaks tailnet SSH, you have locked the door with the keys inside.

Verify both directions

Adapt the identity, define the tag owner, tag the server, and verify SSH succeeds while an unrelated test port remains blocked:

```
ssh ubuntu@lab-server           # should connect
nc -zv -w 3 lab-server 5432     # should time out
```

The second command matters as much as the first. A policy that allows what you want proves nothing until you also confirm it blocks what you did not grant. Port 5432 stands in for anything else running on the server: nothing granted, nothing reachable.

If SSH fails after the change, check the exact spelling of your identity and the tag. A typo in `src` silently matches nobody, and the deny-by-default model does the rest.

Organize people and services

Use groups for people, tags for service devices, autogroups for built-in roles, and tag owners to control classification.

Policies become difficult to review when every rule repeats individual email addresses and IPs.

Three devices and one admin? Raw identities are fine. Ten people and thirty servers? Every hire and every new machine means editing every rule that mentions them. The fix is a layer of names.

Groups for people

Groups collect users. Define them once, reference them everywhere:

```
{
  "groups": {
    "group:operators": ["flavio@example.com", "anna@example.com"]
  }
```

```
}
```

Now a grant can say `"src": ["group:operators"]`. When someone joins the team, you edit one line, and every rule that references the group follows.

Tags for services

Tags describe non-human devices by role, such as `tag:server` or `tag:database`. A tagged device is not owned by whoever enrolled it. Its identity is the tag. This matters for servers: the machine should keep working, and keep its policy identity, after the person who set it up leaves.

`tagOwners` controls who may assign each tag, because assigning a powerful tag changes authorization:

```
{
  "tagOwners": {
    "tag:server": ["group:operators"],
    "tag:database": ["group:operators"]
  }
}
```

If anyone could apply `tag:database` to their own laptop, they would inherit every grant that targets databases. Tag assignment is a privileged operation. Treat the owner list accordingly.

Autogroups for built-in roles

Autogroups select built-in sets such as members or administrators. `autogroup:member` means every user in the tailnet, `autogroup:admin` the admins, `autogroup:self` a device's own user. They cover policies like ”everyone may reach the internal wiki” without maintaining a group by hand.

Set up the lab

Create a `group:operators` group and let only that group own `tag:server`. Keep human laptops user-owned instead of tagging everything. A laptop's identity should be its person. That is what makes user-based rules like ”Flavio may SSH to servers” meaningful when you read them a year from now.

Test policy before trusting it

Add positive and negative policy tests so a future edit cannot silently broaden or remove important access paths.

A valid policy can still grant the wrong access. Policy tests record examples that must continue to pass or fail.

The policy syntax checker catches typos. It cannot catch a rule that is syntactically perfect and grants your intern access to the production database. Tests encode intent, and intent is exactly what syntax checking cannot see.

Write the tests

Tailscale policy files support a `tests` section. Each test names a source, then lists connections that must be allowed and connections that must be denied:

```
{
```

```
  "tests": [
    {
      "src": "flavio@example.com",
      "accept": ["tag:server:22"],
      "deny": ["tag:server:5432"]
    },
    {
      "src": "anna@example.com",
      "deny": ["tag:server:22"]
    }
  ]
}
```

Test that the operator can reach SSH, that an ordinary member cannot reach it, and that neither identity receives an unrelated port. The tests run when you save the policy. A failing test blocks the save, so the bad policy never goes live.

Negative tests carry the weight

The `deny` entries are the valuable ones. An `accept` test fails loudly the moment someone needs the access, because their connection stops working. A missing `deny` fails silently: the unwanted access exists, nobody notices, and it waits there until someone finds it.

Every time you grant something narrow, add a `deny` test for the broader thing you deliberately did not grant.

Break it on purpose

Add at least one expected-accept and one expected-deny case for the lab server, then deliberately break a rule and confirm validation catches it.

Change `tcp:22` to `tcp:*` in your grant and try to save. Your `deny` test for port 5432 should fail and refuse the save. That refusal is the whole point: a future edit, made by a tired version of you six months from now, cannot silently broaden access past what the tests pin down.

Keep tests beside the policy. Review the effective change before saving, especially when changing tags or broad selectors.

Check your understanding: control access

Check grants, deny-by-default behavior, selectors, groups, tags, tag owners, policy tests, and least-privilege changes.

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

Take this interactive quiz online

Services and SSH

Reach private services, configure Tailscale SSH, use Serve, and choose sharing boundaries.

Keep a service private

Bind a small HTTP service to the server and reach it through Tailscale without publishing it on a public DNS record.

A service can remain private even when the server has a public IP. The important boundary is which interface and firewall path accepts the connection.

Private is not a property of the service. It is a property of where the service listens and what can reach it. A web app bound to all interfaces on a public server is public, tailnet or not.

Start a loopback-only service

Run a small development service on the lab server, bound to loopback so nothing outside the machine reaches it directly:

```
python3 -m http.server 3000 --bind 127.0.0.1
```

Binding to `127.0.0.1` is the strictest choice: even the tailnet interface cannot reach it yet. We will use Tailscale Serve to expose this loopback listener privately in a later lesson.

For now, test the middle option: bind to the Tailscale interface directly.

```
tailscale ip -4
# 100.101.9.23
python3 -m http.server 3000 --bind 100.101.9.23
```

The service now listens only on the tailnet address. From your laptop:

```
curl http://lab-server:3000/
```

It answers, and it traveled the private path.

Verify the denial, not just the success

Confirm the service is not reachable through the server's public address:

```
curl -m 5 http://203.0.113.40:3000/
# curl: (28) Connection timed out
```

Do not assume Tailscale automatically closes a service that also listens on a public interface. Tailscale adds a private path; it does not remove public ones. A service bound to `0.0.0.0` listens on every interface, public included, and installing Tailscale changes nothing about that. This is the most common way a ”private” service turns out to be public.

Restrict its tailnet port with a grant too. Interface binding and access policy are separate layers, and you want both: binding controls where the listener exists, the grant controls who inside the tailnet may use it.

Configure Tailscale SSH

Let Tailscale manage SSH authentication on the private interface while retaining a separate tested recovery path during migration.

Tailscale SSH uses tailnet identity and policy instead of distributing a separate user SSH key for each connection.

When it is on, the Tailscale client on the server handles connections to TCP port 22 that arrive

over the tailnet, and it authenticates them with your tailnet identity. No `authorized_keys` files to distribute, no key rotation when someone leaves. Access follows the policy file.

Enable it on the server

Enable it on the destination with `tailscale set --ssh`:

```
sudo tailscale set --ssh
```

One warning before you run it: existing connections to the Tailscale IP can hang when you enable it, because the client takes over port 22 handling on the tailnet interface. Keep your recovery session open, the one on the public address or console, while you flip this switch.

Permit it in policy

The policy must permit both network access to TCP port 22 and an entry in the `ssh` section for the destination and operating-system users:

```
{
  "ssh": [
    {
      "action": "accept",
      "src": ["flavio@example.com"],
      "dst": ["tag:server"],
      "users": ["ubuntu"]
    }
  ]
}
```

The `users` list names the OS accounts the source may become. Granting `root` here is possible and should be rare. Name the specific unprivileged account the work needs.

Test and verify the boundary

```
ssh ubuntu@lab-server
```

The connection authenticates through Tailscale. Identity came from the tailnet, not from a key in `~/.ssh`.

Keep your recovery session open, enable Tailscale SSH, test a new session, then confirm an unapproved source cannot connect. Have a second user try, or temporarily remove your `ssh` rule and watch the connection be rejected.

The detail worth remembering: Tailscale SSH only answers on the tailnet path. SSH to the public address still goes through `sshd` and normal keys. That is your migration safety net, until you deliberately close it.

Require reauthentication for sensitive SSH

Use Tailscale SSH check mode to require a recent identity-provider login before a sensitive administrative connection.

An accepted SSH rule can connect immediately. A `check` rule adds a recent identity check before the connection proceeds.

The threat here is not a broken policy. It is a correct policy plus an unlocked laptop on a café table. Whoever sits down at that keyboard holds a live tailnet identity and every grant that comes with it.

How check mode works

Check mode asks the user to reauthenticate with the identity provider, then remembers that check for the configured period. The first connection opens a browser page. You log in with your identity provider, including any second factor it enforces, and the SSH session then proceeds. Connections within the check period pass without friction.

```
{
  "ssh": [
    {
      "action": "check",
      "checkPeriod": "12h",
      "src": ["flavio@example.com"],
      "dst": ["tag:server"],
      "users": ["ubuntu"]
    }
  ]
}
```

Change `action` from `accept` to `check` and set a `checkPeriod`. Twelve hours means roughly one browser reauthentication per working day.

Test it

Change the practice SSH rule to check mode, use a short documented check period for the lab, and verify the browser reauthentication flow:

```
ssh ubuntu@lab-server
```

```
# Tailscale SSH requires an additional check.
# To authenticate, visit: https://login.tailscale.com/a/9e8f7a6b5c
```

Complete the login and the session opens. Open a second SSH connection right away: no prompt this time, because the check is still fresh.

What it does not do

It reduces the value of an unattended unlocked device. It does not replace operating-system authorization or careful control of allowed SSH usernames. The `users` list still decides which accounts you can become, and what those accounts may do remains the OS's business.

My advice: use `accept` for routine paths and `check` for anything you would call administrative. If typing `sudo` on that box would make you pause, the connection deserves a `check`.

Choose Serve, sharing, or Funnel

Publish a loopback service inside the tailnet with Serve and distinguish it from device sharing and public Internet exposure.

`tailscale serve` exposes local content or a local server to permitted tailnet users over a tailnet HTTPS name.

You built a loopback service in a previous lesson. Serve is the piece that publishes it, privately, with a real HTTPS certificate for the device's `ts.net` name.

Serve: private to the tailnet

```
tailscale serve --bg localhost:3000
tailscale serve status
```

```
https://lab-server.tail4a5b6.ts.net/
|-- proxy http://localhost:3000
```

Serve stays private to the tailnet. Only devices in your tailnet, subject to your policy, can reach that URL. The `--bg` flag keeps the proxy running in the background instead of holding your terminal.

Open the reported HTTPS URL from your laptop. TLS works out of the box because Tailscale provisions a certificate for the device name.

Sharing: one device, one invited user

Sharing gives a specific external Tailscale user access to a device. You send an invite, they accept from their own tailnet, and your policies still control what they can do on it. Good for "my collaborator needs the staging box" without merging two networks.

Funnel: the public Internet

Funnel is different: it publishes a local service to the public Internet. Anyone with the URL can reach it. No tailnet membership, no identity, nothing. Treat enabling Funnel with the same seriousness as opening a port on your router, because that is what it means. If the app behind it has weak or missing authentication, you just published that weakness to the world.

Choose the smallest audience and remember that application authentication may still be required. Private network does not mean logged in.

Practice the full cycle

Enable Serve, open the reported HTTPS URL from your laptop, test a denied user if available, then turn Serve off and confirm the route closes:

```
tailscale serve off
tailscale serve status
# No serve config
```

Publishing should be reversible, and you should know where the off switch is before you need it.

Check your understanding: services and ssh

Check private service exposure, Tailscale SSH, dual authorization rules, check mode, Serve, Funnel boundaries, and sharing choices.

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

Take this interactive quiz online

Subnets and exit nodes

Advertise private routes, approve them, route Internet traffic, and avoid route and DNS surprises.

Understand subnet routers

Use one Tailscale device to route traffic to private resources that cannot run the Tailscale client themselves.

Not every printer, database, appliance, or legacy server can install Tailscale. A subnet router connects an existing IP prefix to the tailnet.

Think of your home NAS, a managed switch's admin page, or an office printer. No Tailscale client exists for them, or policy forbids installing one. Instead, one Linux machine on that network runs Tailscale and forwards traffic on behalf of its neighbors.

How the traffic flows

Tailnet clients send matching destination traffic to the router, which forwards it into the local subnet. Your laptop learns a route: "for `192.168.50.0/24`, send to the subnet router." Packets travel encrypted through the tailnet to the router, then hop onto the physical LAN as ordinary local traffic.

The devices behind the router never know Tailscale exists. By default the router applies source NAT, so the printer sees a connection coming from the router's LAN address, a neighbor it already trusts.

Reachability is not permission

The route extends reachability, not permission: route approval controls whether clients learn the route, while access policy controls who may use it.

Two separate switches. An admin approves the advertised prefix, and a grant decides which users may send traffic into it. Approving the route while forgetting the grant, or the reverse, is the most common "it doesn't work" in this module.

Pick your lab subnet carefully

Choose a disposable private subnet and identify one resource behind it. Then check your own local network before advertising anything:

```
ip -4 addr show
```

```
2: wlan0    inet 192.168.1.34/24 ...
```

Do not advertise a prefix that overlaps your current local network during the first test. If your laptop sits on `192.168.1.0/24` and a remote site advertises the same prefix, the client cannot tell local neighbors from routed destinations. The classic symptom: local printers vanish while the remote subnet "works". Pick something like `192.168.50.0/24` that appears nowhere else in your life.

Advertise and approve a subnet route

Enable IP forwarding, advertise one narrow private prefix, approve it, grant access, and verify the routed destination.

A Linux subnet router must forward packets between its Tailscale and local interfaces.

That is an OS-level capability, and it is off by default. Without it, the route can be advertised, approved, and granted, and every packet still dies inside the router.

Enable IP forwarding

Enable forwarding using the current Tailscale instructions:

```
echo 'net.ipv4.ip_forward = 1' | sudo tee -a /etc/sysctl.d/99-tailscale.conf
echo 'net.ipv6.conf.all.forwarding = 1' | sudo tee -a /etc/sysctl.d/99-tailscale.conf
sudo sysctl -p /etc/sysctl.d/99-tailscale.conf
```

This persists across reboots because it lives in a sysctl config file, not just in the running kernel.

Advertise the route

Then advertise the exact CIDR with `tailscale set`:

```
sudo tailscale set --advertise-routes=192.168.50.0/24
tailscale status
```

Advertise the narrowest prefix that covers what you need. One database server needs its `/32`, not the whole office network. Approve the route in the admin console unless an `autoApprovers` rule covers it. Advertising is a request; approval is the admin's decision. Once approved, the machine shows a ”Subnets” badge in the console.

Add an access grant for the destination prefix before testing:

```
{
  "src": ["flavio@example.com"],
  "dst": ["192.168.50.0/24"],
  "ip":  ["tcp:80", "tcp:443"]
}
```

Verify from a client

Linux clients do not accept advertised routes automatically. On your Linux laptop:

```
sudo tailscale set --accept-routes
curl -m 5 http://192.168.50.20/
```

Reach one private resource through the route, inspect its observed source address, and document that subnet routing uses SNAT by default. In the destination's logs you will see the router's LAN address, not your laptop's tailnet address. That default keeps LAN devices happy, since replies go to a local neighbor, but it hides the real client. Write it down for whoever reads those logs later.

If the connection times out, work the chain in order: forwarding enabled, route advertised, route approved, grant present, `--accept-routes` set on the client.

Understand exit nodes

Compare an ordinary overlay path with an exit node that routes public Internet traffic through another tailnet device.

Tailscale normally routes only tailnet and approved subnet destinations. Your ordinary Internet traffic keeps using the local connection.

That default is worth appreciating. The tailnet is an overlay for private traffic, not a middleman for everything. Browsing, streaming, and system updates never touch it.

An exit node changes that deal.

What an exit node does

An exit node advertises the IPv4 and IPv6 default routes. A client explicitly chooses that node, then sends Internet traffic through it. All of it: every website, every API call leaves the Internet from the exit node's location, encrypted along the way from your device to that node.

Both halves matter. Advertising makes the offer; nothing routes through the node until a client opts in. And the opt-in is per client. Choosing an exit node on your laptop changes nothing for your phone.

You can list what your tailnet offers:

```
tailscale exit-node list
```

```
IP              HOSTNAME     COUNTRY   CITY      STATUS
100.101.9.23    lab-server   Italy     Milan     -
```

What actually changes

This changes egress location and trust, adds latency, and can affect local-network access.

Websites see the exit node's public IP. On hostile Wi-Fi, the local network sees only encrypted tunnel traffic. The exit node's operator and network now sit in your Internet path, so you are moving trust, not removing it. Every packet takes a detour, and a distant exit node adds real milliseconds to everything. Traffic to your local printer or NAS may stop working unless the client explicitly allows LAN access alongside the exit node.

Exit nodes are available on all plans.

When you need one

Good reasons: an untrusted network you want to tunnel out of, or needing to appear from the exit node's network.

Write down when you need private device access and when you truly need different Internet egress. Do not enable an exit node to solve a DNS-only problem. If names resolve wrong on some network, fix the DNS configuration. Rerouting your entire Internet connection is a heavy tool for a light problem.

Configure and test an exit node

Advertise a disposable Linux exit node, approve it, select it from a client, and verify both egress and recovery behavior.

The exit node needs packet forwarding and permission to advertise default routes. The client then opts into using it.

The forwarding requirement is the same one a subnet router has. If you completed that lesson on this machine, it is already done. Otherwise enable IPv4 and IPv6 forwarding through sysctl first.

Advertise and approve

Advertise exit-node capability on the server, approve it in the admin console, and select it on the laptop:

```
sudo tailscale set --advertise-exit-node
```

The machine gets an "Exit node" badge in the console once an admin approves it. Until then, clients cannot select it.

Select it and measure the change

Record the public IP before and after:

```
curl -4 https://ifconfig.me
# 93.44.120.15        <- your normal egress
tailscale set --exit-node=lab-server
curl -4 https://ifconfig.me
# 203.0.113.40        <- the exit node's public IP
```

The egress address changed. Your Internet traffic now leaves from the server's network.

Test whether local LAN access is allowed. By default, using an exit node cuts you off from your local network. If you still need the printer or NAS while tunneled:

```
tailscale set --exit-node=lab-server --exit-node-allow-lan-access
```

Keep the way out ready

Keep a command ready to stop using the node if routing or DNS fails:

```
tailscale set --exit-node=
```

The empty value clears the selection and ordinary routing returns. Learn this command before you need it. If the exit node's connectivity or DNS breaks while selected, your whole Internet connection breaks with it, and you will be typing this without a search engine to help.

Verify the egress address changes, test one tailnet destination, then disable the exit node and confirm ordinary routing returns. The tailnet check matters: private connectivity should keep working while the exit node handles public traffic.

Check your understanding: subnets and exit nodes

Check subnet routing, forwarding, route advertisement and approval, access rules, SNAT, exit nodes, DNS, and safe verification.

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

Take this interactive quiz online

Automation and device trust

Enroll servers safely with auth keys, tags, expiry, approval, OAuth clients, and revocation.

Choose an authentication method

Use interactive login for people and narrowly scoped auth keys or OAuth clients for servers and automated environments.

A person can open a browser and authenticate. An unattended server, container, or CI runner needs a non-interactive enrollment path.

Interactive login is the right default for anything with a human at the keyboard. The identity provider does the work, second factors apply, and the resulting node belongs to that person. The interesting decisions start when nobody is there to click.

The non-interactive options

A one-off auth key is the safest default for one server. It enrolls exactly one node, then becomes useless. Stolen after use, it is worth nothing.

```
sudo tailscale up --auth-key=tskey-auth-kFJb851CNTRL-example
```

Reusable keys can enroll multiple nodes but are dangerous if stolen. Anyone holding one can add devices to your tailnet until it expires or you revoke it. If you must use one, give it a short expiry and treat it like a root password.

Ephemeral nodes disappear after going offline and fit disposable jobs. A CI runner that lives for one build has no business leaving a permanent tailnet entry behind. Create the key with the ephemeral option and cleanup happens on its own.

Tags assign service identity during enrollment rather than attaching the node to a person. Combine them with any key type: the node's identity becomes `tag:ci` or `tag:server`, and policy targets the role.

Classify your fleet

Classify a laptop, long-lived server, autoscaled runner, and CI job. Choose interactive login, one-off key, reusable key, or ephemeral enrollment for each.

My answers: laptop, interactive login. Long-lived server, one-off key with a tag. Autoscaled runners, ephemeral keys minted by automation, which is the next lessons' topic. CI job, ephemeral tagged key, so each run enrolls fresh and vanishes after.

The failure mode to avoid: one reusable key pasted into five places ”temporarily”. Six months later nobody remembers where, and revoking it breaks something unknown. Choose the key type that matches the machine's lifespan.

Enroll a tagged server safely

Create a short-lived one-off key with a permitted tag, inject it without logging, and revoke it after enrollment.

Auth keys are credentials. Treat the value like a password even when it can only be used once.

An unused one-off key in the wrong hands enrolls a device into your tailnet carrying whatever tags it was created with. That device inherits every grant targeting those tags. The window may be small; the blast radius is not.

Prepare before generating

Define the tag owner first, generate a short-lived key with the `tag:server` tag, and store it in the deployment platform's secret channel.

The order matters. `tagOwners` must permit the tag before a key can carry it. Generate the key in the admin console under Settings, Keys: one-off, short expiry (an hour is plenty for one deliberate enrollment), with `tag:server` pre-applied.

Store the value where secrets live in your platform: a secret manager, a CI secret, or an environment variable injected at deploy time. Not in a repo, not in a Dockerfile, not in a wiki page.

Enroll without leaking

Pass it to `tailscale up` without committing or printing it:

```
sudo tailscale up --auth-key="$TS_AUTH_KEY" --advertise-tags=tag:server
```

The key arrives through an environment variable, so the literal value never lands in shell history. Resist running `echo $TS_AUTH_KEY` to "check it". That is printing a password.

Verify, then clean up

Use a disposable key, verify the node is tagged rather than user-owned, clear the shell value, and revoke the key from the admin console.

```
tailscale status
# 100.101.9.23  lab-server  tagged-devices  linux  -
unset TS_AUTH_KEY
```

The owner column reading `tagged-devices` instead of your email is the confirmation that matters. This node's identity is now its role.

Revoke or delete the key when the enrollment is complete. Yes, even a spent one-off key: removing it costs nothing and keeps the key list meaningful. A tidy key list is the difference between noticing a rogue key and scrolling past it.

Use OAuth clients for repeated automation

Replace a long-lived reusable auth key with an OAuth client that creates short-lived, scoped auth keys when needed.

Repeated automation should not distribute one reusable device-enrollment credential to every environment.

A reusable auth key in five CI pipelines is five copies of the same secret, one expiry date, and no way to tell which pipeline enrolled which node. When it leaks, and shared long-lived secrets do leak, you rotate everything at once, in a hurry.

What an OAuth client changes

A Tailscale OAuth client receives limited scopes and tags. Automation uses its client secret to request fresh auth keys, then enrolls nodes with those short-lived keys.

Create the client in the admin console with only the auth keys scope and only the tags it may hand out, such as `tag:ci`. The client can mint keys, but never keys with broader power than its own

scope and tags. That bound is the security win: a compromised CI system can enroll `tag:ci` nodes, not `tag:server` ones.

The minting flow

The automation exchanges its credentials for an API access token:

```
curl -s -d "client_id=$TS_CLIENT_ID" -d "client_secret=$TS_CLIENT_SECRET" \
  https://api.tailscale.com/api/v2/oauth/token
```

With the returned token, it calls the keys API to create a one-off, ephemeral, pre-tagged auth key, and passes that to enrollment:

```
sudo tailscale up --auth-key="$TS_MINTED_KEY" --advertise-tags=tag:ci
```

Each job gets its own key, used once, expiring fast. Nothing long-lived ever touches a runner.

Keep the OAuth secret in a vault, scope tags narrowly, and rotate the client if the secret is exposed. The secret is powerful, but it lives in exactly one place, your secret manager, instead of being copied around.

Design the lab flow

Design an enrollment flow for CI runners that creates one ephemeral tagged node per job and removes access when the job ends. With ephemeral keys, removal is free: the node evaporates once it goes offline. The audit story improves too, because every enrollment traces back to one client and one job.

Manage expiry and device approval

Use node-key expiry, device approval, and deliberate exceptions to balance safe rotation with reliable infrastructure access.

Node keys expire independently from the auth key used to enroll a device. Expiry forces periodic reauthentication and key rotation.

The auth key was the ticket in the door. The node key is the device's ongoing WireGuard identity, and it expires on a schedule (180 days by default). When it does, the device drops out of the tailnet until someone reauthenticates:

```
sudo tailscale up --force-reauth
```

That is a feature for laptops and phones. A device that left the company, or a tablet lost on a train, stops being a tailnet member on a schedule even if nobody remembers to remove it.

Where expiry hurts

Tagged infrastructure normally has key expiry disabled, so protect tags, monitor ownership, and revoke lost or replaced devices promptly.

Nobody wants the production subnet router dropping offline at day 180, waiting for a browser login. The trade is explicit: the machine never expires, so humans must do the reviewing that expiry would have forced. Check last-seen times, retire replaced hardware, and treat the machines list as an inventory you own. You can also disable expiry per device in the admin console. Reserve that for infrastructure, never for personal devices.

Device approval

Device approval adds an administrator check before a new device can exchange tailnet traffic. Enable it under Settings, Device management, and every new enrollment waits until an admin reviews it. Valid credentials stop being sufficient; a human confirms the device belongs.

Preapproved auth keys can automate trusted builds. A key created with preapproval skips the queue, which is right for infrastructure enrolled through your deployment pipeline, where the approval effectively happened when someone created the key.

Practice the lifecycle

Enable device approval in a disposable tailnet if practical, enroll one waiting device, review its identity, approve it, then revoke it and confirm traffic stops:

```
tailscale ping lab-tablet
# succeeds while approved, fails after revocation
```

Revocation is the step nobody rehearses. Do it once calmly, so the first real one is not also the first attempt.

Check your understanding: automation and device trust

Check auth-key types, tags, ephemeral nodes, preapproval, OAuth clients, key expiry, device approval, secrets, and revocation.

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

Take this interactive quiz online

Operate and choose

Diagnose paths, maintain the tailnet, respond to device loss, and choose when Tailscale fits.

Troubleshoot by layer

Diagnose identity, device state, name resolution, policy, route selection, and application behavior in a repeatable order.

"Tailscale is down" is not a useful diagnosis. The failure can live at several independent layers.

Confirm both devices are connected, then test the Tailscale IP, MagicDNS name, policy, subnet or exit route, destination port, and application. `tailscale status` shows peers, `tailscale ping` explains the path, and `tailscale netcheck` reports local network conditions.

```
tailscale status
tailscale ping lab-server
tailscale netcheck
```

Break one layer at a time—name, grant, listener, or route—and record which command identifies the failure without changing unrelated settings.

Test names, reachability, and the Tailscale path separately:

```
tailscale status
```

```
tailscale ping server-name
ping -c 3 server-name
tailscale netcheck
```

tailscale ping tells you whether the peers can establish a Tailscale path and whether it is direct or relayed. Ordinary ping also depends on the destination operating system and firewall. If the name fails, retry with the Tailscale IP before changing ACLs or routes.

Interpret direct and relayed performance

Use path information and network checks to explain latency or throughput without treating every DERP connection as a security failure.

A connection can be correct and encrypted while performing worse than expected because it uses a relay.

Before touching anything, find out which path you are on. Guessing about performance without path information wastes everyone's time.

Read the path

`tailscale ping` reports whether the peer is reached directly or through a DERP region:

```
tailscale ping lab-server
```

```
pong from lab-server (100.101.9.23) via DERP(fra) in 61ms
pong from lab-server (100.101.9.23) via 203.0.113.40:41641 in 14ms
```

This output shows an upgrade happening: the first reply relays through Frankfurt, then the direct path finishes negotiating and latency drops. If every reply stays on DERP, the peers could not establish a direct route.

`tailscale netcheck` helps reveal UDP and mapping behavior:

```
tailscale netcheck
```

```
* UDP: true
* MappingVariesByDestIP: true
* Nearest DERP: Frankfurt
```

`UDP: false` means outbound UDP is blocked, so everything relays, full stop. `MappingVariesByDestIP: true` indicates a hard NAT, which makes direct paths much harder to establish.

Interpret before acting

A relay is a designed fallback, not plaintext transport. The payload is WireGuard-encrypted end to end; DERP moves ciphertext it cannot read. So the question is never ”is this insecure?”. The question is ”is this fast enough for the workload?”

An SSH session over DERP at 60ms is fine. Nightly multi-gigabyte backups through a relay are not. Improve firewall or NAT conditions only when performance justifies the change, typically by allowing outbound UDP or the documented Tailscale port on the stricter network.

Compare two networks

Compare path, latency, and transfer behavior on two networks if available. Run the same `tailscale ping` from home and from a café, and note how the same peer can be direct from one and relayed from the other. The path is a property of both networks, not of the tailnet.

Keep the explanation separate from assumptions about application speed. A slow app on a direct 8ms path is an application problem. Measure first.

Operate the tailnet safely

Review users, devices, routes, tags, keys, updates, logs, and recovery paths as a recurring maintenance routine.

A private network accumulates old devices and permissions unless someone deliberately removes them.

Nothing rots loudly. The old intern's laptop stays enrolled, a test route stays approved, a forgotten reusable key stays valid. Each is harmless until one becomes the way in. Operating a tailnet is mostly scheduled subtraction.

The monthly review

Review device ownership and last-seen times, delete retired nodes, revoke unused auth and OAuth credentials, inspect advertised routes, update clients, and test important policy paths.

Concretely: walk the admin console machines list sorted by last seen, and question anything silent for a month. Check the keys page for auth keys and OAuth clients you cannot explain. Look at every approved subnet route and exit node and ask whether it is still needed. Routes are easy to approve and easier to forget.

Client versions matter too:

```
tailscale version
# 1.86.2
```

Old clients miss security fixes and path-negotiation improvements. After updating, re-run your policy tests and one real check of the critical path:

```
tailscale ping lab-server
ssh ubuntu@lab-server 'echo ok'
```

Keep a recovery path outside Tailscale

Keep a recovery method outside Tailscale for critical routers and servers. If Tailscale SSH is your only access and a bad policy edit locks you out, you cannot fix the policy that locked you out. A cloud console, a physical keyboard, or classic SSH on a tightly firewalled public port: something that does not depend on the system you are administering.

The lost-device runbook

Respond to device loss by revoking access first. Not ”look for the laptop”, not ”ask around”. Revoke, then investigate. A revoked device can be re-enrolled in minutes if it turns up in a jacket pocket. The reverse mistake has no undo.

Write a monthly checklist and a lost-device runbook. Include who can approve devices, edit policy, advertise routes, and recover a locked-out server. Names, not roles. A runbook that says ”an admin” helps nobody at 2am.

Choose when Tailscale fits

Compare Tailscale with a traditional VPN, plain WireGuard, public HTTPS, and application-level access before choosing the architecture.

Tailscale is excellent for private device and service connectivity, but it is not the answer to every networking problem.

After this course you will be tempted to route everything through the tailnet. Resist that. The tool earned your trust for one class of problems. Keep it there.

Where it wins

Choose it when identity-aware private networking, NAT traversal, stable names, and managed policy remove meaningful work.

The pattern: a defined set of people and machines that need to reach each other privately, across networks you do not control. SSH to servers, internal dashboards, databases behind subnet routers, a home lab reachable from anywhere. One policy file, no port forwarding, names that survive reboots and moves.

Where something else wins

Choose plain WireGuard when you need minimal self-managed peers. Two static servers with fixed public IPs do not need a coordination plane. A short WireGuard config connects them with zero external dependencies. You give up NAT traversal, key distribution, and policy management, and with two stable peers there is little to give up.

Use public HTTPS for public users. Customers will not install a VPN client to reach your product. If the audience is ”anyone”, the answer is a normal public deployment with TLS, not Funnel as a production strategy.

And keep application authentication even when the network is private. The tailnet decides which machines may talk. It does not know which humans should see which records. A database that trusts every tailnet connection has an authorization model exactly one stolen laptop wide.

The review exercise

Review three systems you use. For each, choose public Internet, Tailscale node access, subnet routing, exit-node routing, or another design, and name the trust tradeoff.

Naming the tradeoff is the real test. ”Tailscale node access: I trust the tailnet's device list and my grants instead of a public firewall” is a decision. ”Tailscale because it's what we use” is a habit. Write the sentence for each system, and keep it where the next person will find it.

Check your understanding: operate and choose

Check layered troubleshooting, direct and relayed paths, routes, DNS, policy, updates, audit, device loss, cleanup, and architecture choices.

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/tailscale/.