SSH Course

Flavio Copes

Updated August 3, 2026

Contents

Use SSH safely for remote access, host verification, keys, file transfer, tunnels, jump hosts, automation, and recovery.

What you'll learn: Create and operate a verified path to an Ubuntu server with personal keys, narrow tunnels and automation, rotation, and a tested recovery runbook.

Connections and trust

Understand the handshake, verify host identity, and diagnose explicit connections.

Understand the SSH connection

Trace client, server, transport encryption, host authentication, user authentication, and the remote session.

Before choosing a tool or writing more code, make the requirement concrete. The local client verifies the VPS host key before the server checks whether the operator may log in.

SSH protects a client-server connection and separates proof of the server from proof of the user. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to collapse host trust and user login into one vague idea of "the key worked". The happy path may still work, which makes the mistake easy to miss. draw the handshake in order and identify the credential or key used at each trust decision. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Connect to the intended host and know exactly which identity, network path, and trust decision made it possible. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run a verbose connection to the disposable server and annotate the lines for negotiation, host verification, and user authentication. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Verify the host key

Confirm a new server fingerprint through an independent channel before accepting it.

Start from the behavior you can observe. The VPS console or provider metadata supplies a fingerprint that can be compared out of band.

The first-connection prompt cannot prove the key is correct by itself because the network presenting the host also presents the fingerprint. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to type yes automatically because the hostname looks familiar. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to obtain the expected fingerprint independently, compare it exactly, and record the verified host identity. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Connect to the intended host and know exactly which identity, network path, and trust decision made it possible. Record enough evidence that another developer can repeat the result without relying on your memory.

Create a new server, collect the fingerprint from its console, and verify it before the first SSH acceptance. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Understand known_hosts

Read, update, hash, and troubleshoot stored host keys without deleting trust data blindly.

A rebuilt disposable server has a legitimately new key, but the operator confirms the rebuild before removing only the old entry. That contrast gives us something useful to test instead of a rule to memorize.

`known_hosts` records host identities so a future change becomes a loud security event. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can delete the entire known_hosts file whenever one host changes and still get one successful demo. Push past the demo. locate the exact matching entry, investigate why it changed, verify the replacement, and update only that host, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Connect to the intended host and know exactly which identity, network path, and trust decision made it possible. Save the before-and-after evidence now; it will make the final project review much more honest.

Trigger a controlled host-key mismatch and document the investigation and narrow repair. Save the evidence, then explain which requirement would force you to choose a different design.

Connect with explicit options

Specify user, host, port, and identity clearly and use verbose modes to diagnose selection.

The first test names the operator user, server address, nondefault port if any, and intended identity file explicitly. This is a small part of a safe operator path from a local machine to a disposable Ubuntu server, including file transfer, tunnels, automation, and recovery, but the boundary it creates affects everything that follows.

SSH combines command-line options, per-host configuration, wildcard configuration, and system defaults using precedence rules. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to change several client and server settings at once until a connection succeeds. It makes later failures harder to locate. Instead, start with an explicit command, use effective-configuration and verbose output, and change one diagnosed boundary at a time. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Connect to the intended host and know exactly which identity, network path, and trust decision made it possible. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Connect with one intentionally wrong user, port, and identity in separate attempts and identify each failure layer. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: connections and trust

Check SSH roles, transport, host identity, knownhosts, first connection, users, ports, and evidence.

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

Take this interactive quiz online

Keys, agent, and config

Create protected identities, install public keys, and write maintainable client config.

Generate a dedicated key

Create a modern key pair with a useful comment and a purpose-specific filename.

Before choosing a tool or writing more code, make the requirement concrete. The VPS operator key is distinct from unrelated Git hosting and automation identities.

A key pair has a private signing key and a public key that servers may authorize; the private half stays on the client. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to reuse one unnamed private key for every service and device. The happy path may still work, which makes the mistake easy to miss. generate a dedicated supported key, label its purpose, protect the file, and back it up only according to policy. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Create a maintainable client identity setup without copying private keys or making authentication implicit. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Generate the key, inspect its fingerprint and public half, and prove the private half never leaves the client. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Protect private keys and use the agent

Use passphrases and an agent to balance theft resistance with usable sessions.

Start from the behavior you can observe. The operator unlocks the key locally and verifies which identities the agent currently offers.

A passphrase protects a copied private-key file, while an agent can hold an unlocked key for a controlled session. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to remove the passphrase or forward the agent everywhere for convenience. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to use a strong passphrase, load only needed identities, bound agent lifetime, and avoid forwarding unless the threat model requires it. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Create a maintainable client identity setup without copying private keys or making authentication implicit. Record enough evidence that another developer can repeat the result without relying on your memory.

List agent identities, connect with identities-only behavior, remove the key, and observe the authentication difference. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Install and restrict public keys

Add the correct public key to the correct account and understand authorizedkeys restrictions.

A deployment identity can run one controlled command without receiving an unrestricted interactive shell. That contrast gives us something useful to test instead of a rule to memorize.

`authorized_keys` grants login capability to a user account and can constrain source addresses, commands, forwarding, or other features. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can paste private material onto the server or append keys without knowing which account owns the file and still get one successful demo. Push past the demo. install only the public line, set correct ownership and modes, and add restrictions for narrow automation identities, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Create a maintainable client identity setup without copying private keys or making authentication implicit. Save the before-and-after evidence now; it will make the final project review much more honest.

Create a restricted test key and verify both the allowed operation and a denied interactive shell. Save the evidence, then explain which requirement would force you to choose a different design.

Write maintainable SSH config

Use host aliases and narrowly scoped options to make the intended connection repeatable.

The alias `status-vps` declares host name, user, identity, and identities-only behavior without affecting other hosts. This is a small part of a safe operator path from a local machine to a disposable Ubuntu server, including file transfer, tunnels, automation, and recovery, but the boundary it creates affects everything that follows.

Client configuration reduces typing, but broad wildcard rules can silently select the wrong user, key, proxy, or forwarding behavior. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to put sensitive or host-specific options under an unrestricted wildcard. It makes later failures harder to locate. Instead, create specific host blocks, inspect effective configuration, and keep defaults conservative. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Create a maintainable client identity setup without copying private keys or making authentication implicit. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Build two aliases for the same server with different safe purposes and compare their effective configuration. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: keys, agent, and config

Check key generation, passphrases, public-key installation, agent use, config aliases, identities, and rotation.

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

Take this interactive quiz online

Files, tunnels, and jumps

Run commands, transfer files, forward ports, and cross bastions safely.

Run remote commands

Execute one quoted remote command and distinguish local expansion from remote shell interpretation.

Before choosing a tool or writing more code, make the requirement concrete. A diagnostic command uses explicit quoting and an absolute path rather than depending on an interactive profile.

The local shell parses the SSH command first, then the remote side runs the provided command according to its shell and account environment. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to insert untrusted values into a remote shell command or assume every quote reaches the remote unchanged. The happy path may still work, which makes the mistake easy to miss. minimize shell interpolation, pass fixed commands, and inspect both local and remote parsing boundaries. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Use SSH as a controlled transport for commands, files, and connections while understanding each new exposure. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run commands containing spaces, variables, and redirection and explain which machine interprets each token. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Transfer files with scp and sftp

Choose batch copy or interactive file transfer and verify direction, destination, permissions, and integrity.

Start from the behavior you can observe. A release archive uploads to a staging filename, is checksummed, then is moved into place deliberately.

File transfer over SSH inherits host and user authentication but still needs careful path and overwrite handling. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to reverse source and destination or overwrite production paths without inspection. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to use explicit paths, test with a disposable destination, verify size or checksum, and set final ownership intentionally. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Use SSH as a controlled transport for commands, files, and connections while understanding each new exposure. Record enough evidence that another developer can repeat the result without relying on your memory.

Upload and download a test archive, compare checksums, and verify the resulting remote permissions. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Create local and remote tunnels

Map listening and destination endpoints for local and remote port forwarding before opening them.

A local-only database port becomes available on `127.0.0.1` of the operator machine without making the database public. That contrast gives us something useful to test instead of a rule to memorize.

Local forwarding exposes a remote-side destination to a local listener; remote forwarding exposes a client-side destination from the server side. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can copy a forwarding flag without drawing which interface listens on which machine and still get one successful demo. Push past the demo. write all four endpoints, bind narrowly, disable an unnecessary remote command, and test who can actually connect, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Use SSH as a controlled transport for commands, files, and connections while understanding each new exposure. Save the before-and-after evidence now; it will make the final project review much more honest.

Build a local tunnel to a private test service and prove it is unavailable from another machine. Save the evidence, then explain which requirement would force you to choose a different design.

Use jump hosts without spreading keys

Reach a private host through a bastion while keeping authentication end to end where possible.

The client authenticates to the bastion and then to the private server using local identities and host verification for both. This is a small part of a safe operator path from a local machine to a disposable Ubuntu server, including file transfer, tunnels, automation, and recovery, but the boundary it creates affects everything that follows.

ProxyJump forwards the connection path through an intermediate host without requiring a private key to live on that host. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to copy private keys to the bastion or enable agent forwarding by habit. It makes later failures harder to locate. Instead, configure ProxyJump, verify every host key separately, and use agent forwarding only after reviewing bastion risk. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Use SSH as a controlled transport for commands, files, and connections while understanding each new exposure. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Connect to a private disposable host through a bastion and show where each key operation occurs. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: files, tunnels, and jumps

Check remote commands, scp and sftp, local and remote forwarding, jump hosts, agent forwarding, and exposure.

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

Take this interactive quiz online

Server access and hardening

Use personal accounts, change sshd safely, reduce exposure, and read logs.

Use personal accounts and sudo

Log in as an attributable unprivileged user and elevate only the commands that need it.

Before choosing a tool or writing more code, make the requirement concrete. Each operator has an individual account and key; administrative actions appear in sudo and system logs.

Shared root login hides accountability and increases the impact of a stolen credential. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to share one root key among operators for convenience. The happy path may still work, which makes the mistake easy to miss. create personal accounts, grant the minimum sudo access, and remove access per person without rotating everyone. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Reduce SSH exposure on the server while preserving a tested recovery path before removing existing access. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Create two operator accounts and verify their login, sudo boundary, logs, and independent revocation. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Change sshd configuration safely

Validate configuration and keep a second tested session open before reloading the daemon.

Start from the behavior you can observe. The operator validates the effective daemon configuration, keeps the current session, reloads, and tests a new session before closing anything.

A syntax-valid configuration can still lock you out because keys, users, network policy, or included files behave differently than expected. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to restart the daemon from the only session after several untested changes. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to change one policy, validate it, reload rather than blindly restart, and prove a second login before disconnecting. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Reduce SSH exposure on the server while preserving a tested recovery path before removing existing access. Record enough evidence that another developer can repeat the result without relying on your memory.

Apply a harmless configuration change using the complete safe sequence and record the rollback command. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Reduce authentication and network exposure

Disable unused login methods and restrict who and where can reach SSH according to the real recovery plan.

Password login and direct root login are disabled after personal key access and provider-console recovery are tested. That contrast gives us something useful to test instead of a rule to memorize.

Hardening removes unnecessary paths only after a working replacement and recovery channel are proven. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can disable methods from a checklist before verifying keys, console access, and automation dependencies and still get one successful demo. Push past the demo. inventory users and clients, prove key access and recovery, then remove unused authentication and network exposure, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Reduce SSH exposure on the server while preserving a tested recovery path before removing existing access. Save the before-and-after evidence now; it will make the final project review much more honest.

Test allowed and denied users, password attempts, root attempts, and a firewall rule from a separate session. Save the evidence, then explain which requirement would force you to choose a different design.

Read SSH logs and failures

Correlate client verbose output with server authentication logs and network evidence.

The operator distinguishes timeout, host-key failure, rejected key, invalid account, file-mode rejection, and shell startup failure. This is a small part of a safe operator path from a local machine to a disposable Ubuntu server, including file transfer, tunnels, automation, and recovery, but the boundary it creates affects everything that follows.

Client and server logs describe different sides of negotiation, key selection, policy, permissions, and session startup. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to regenerate keys immediately whenever login fails. It makes later failures harder to locate. Instead, locate the failure layer first, compare both sides, and change only the setting supported by evidence. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Reduce SSH exposure on the server while preserving a tested recovery path before removing existing access. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Create four controlled failures and write a diagnosis from logs before applying any fix. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: server access and hardening

Check user accounts, sudo, daemon configuration, authentication methods, network exposure, logs, and lockout prevention.

Answer each question before continuing. Read the explanation after every answer, including the

ones you get right.

Take this interactive quiz online

Automation, rotation, and recovery

Build narrow automation, rotate and revoke keys, respond, and rehearse recovery.

Build narrow automation

Give deployment jobs a dedicated identity, known host, restricted command, and minimal server permissions.

Before choosing a tool or writing more code, make the requirement concrete. The deploy key can invoke one server-side release script and cannot open a shell or forward ports.

Noninteractive SSH removes a human checkpoint, so its allowed action and failure behavior must be narrower and more observable. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to reuse a personal administrator key in CI and disable host checking to make it work. The happy path may still work, which makes the mistake easy to miss. create a dedicated identity, pin the host, restrict authorized_keys, scope server permissions, and log every run. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Operate SSH access over time with narrow automation, regular rotation, revocation, and rehearsed recovery. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run the allowed deployment and attempt a shell, arbitrary command, and tunnel; only the intended action should work. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Rotate and revoke keys

Inventory authorized identities, add and prove replacements, then remove old keys without an outage.

Start from the behavior you can observe. A new operator key is installed and tested before the old fingerprint is removed from every server.

SSH keys do not expire automatically in ordinary authorized_keys setups, so ownership and rotation need an operational process. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to remove the only working key first or leave departed users authorized indefinitely. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to maintain an owner-and-purpose inventory, add and verify the replacement, revoke the old key, and audit all targets. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Operate SSH access over time with narrow automation, regular rotation, revocation, and rehearsed recovery. Record enough evidence that another developer can repeat the result without relying on your memory.

Rotate one test identity across two servers and prove the old key is rejected everywhere. Repeat it

once with an invalid or hostile condition and write down the boundary the failure revealed.

Respond to a lost key or host change

Distinguish a compromised user identity from a changed server identity and respond to the right trust boundary.

The incident plan records affected accounts, servers, logs, replacement keys, and evidence of unauthorized use. That contrast gives us something useful to test instead of a rule to memorize.

A lost private key requires revoking that public key; an unexpected host key requires stopping and investigating the server path. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can clear warnings and issue new keys without determining which side may be compromised and still get one successful demo. Push past the demo. contain the affected identity, preserve evidence, verify infrastructure through an independent channel, rotate, and monitor, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Operate SSH access over time with narrow automation, regular rotation, revocation, and rehearsed recovery. Save the before-and-after evidence now; it will make the final project review much more honest.

Run tabletop exercises for a stolen laptop and an unexpected host-key warning and compare the actions. Save the evidence, then explain which requirement would force you to choose a different design.

Finish the operator runbook

Document connection, transfer, tunnel, hardening, automation, rotation, failure diagnosis, and break-glass recovery.

The runbook names key fingerprints and procedures but never embeds private keys, tokens, or passphrases. This is a small part of a safe operator path from a local machine to a disposable Ubuntu server, including file transfer, tunnels, automation, and recovery, but the boundary it creates affects everything that follows.

Access is reliable when another authorized operator can follow a tested runbook without inheriting secret material. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to write only the happy-path ssh command and call the system maintained. It makes later failures harder to locate. Instead, record prerequisites, trust verification, expected evidence, rollback, revocation, and recovery for every operation. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Operate SSH access over time with narrow automation, regular rotation, revocation, and rehearsed recovery. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Rebuild the disposable server and have a second operator restore safe access from the runbook. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: automation, rotation, and recovery

Check restricted automation, host checking, key rotation, incident response, break-glass recovery, and final verification.

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/ssh/.