Telnet Course

Flavio Copes

Updated August 3, 2026

Contents

Learn Telnet as a readable application protocol, inspect its commands and option negotiation, use a client to explore text protocols, and understand why SSH replaced it.

What you'll learn: Decode a Telnet session, run a safe local lab, inspect HTTP and SMTP conversations over TCP, and choose safer tools for real remote access.

How Telnet works

Understand the TCP connection, Network Virtual Terminal, byte stream, commands, and port 23.

What Telnet is

Understand Telnet as a protocol for bidirectional terminal communication, not only as the command that opens a TCP connection.

Telnet is a protocol for bidirectional, byte-oriented communication between terminals and terminal-oriented processes. It is one of the oldest protocols still documented on the internet: the current specification, RFC 854, dates to 1983, and the idea is older than that.

The job it was built for: you sit at a terminal in one place, and a computer you want to use lives somewhere else. Telnet carries your keystrokes to that machine and its output back to your screen, over a TCP connection.

The word **Telnet** is also used for client programs that implement the protocol. On most systems that program is literally called `telnet`:

```
telnet 192.0.2.20
```

This double meaning causes real confusion, so let's settle it now. This distinction matters. You can use a Telnet client to connect to a different text protocol, but that does not turn the other protocol into Telnet. When someone says ”just telnet to port 80”, they mean using the client as a generic tool. The web server on the other side has never heard of Telnet.

What the protocol provides

Telnet was designed for interactive remote terminals, connecting machines from different vendors that shared almost nothing. It provides three things: a common terminal model, so two different systems agree on what a ”terminal” is; control commands, like interrupting a running process; and a way to negotiate optional behavior between two different systems, so capable clients and servers can agree on extras without breaking simple ones.

Each of these gets its own lesson in this module. Together they are the actual protocol, as opposed to the raw TCP connection underneath.

Why learn it in 2026

Nobody should use Telnet to log into machines anymore, and this course will be blunt about why: everything travels unencrypted, and SSH replaced it decades ago.

But as a learning tool, Telnet is hard to beat. The protocol is readable enough to study by hand. You can type it, read its responses, and watch its control bytes in a hex dump. It is therefore a useful way to learn how application protocols sit inside a TCP connection, even though Telnet is not a safe choice for remote login on an untrusted network.

Learn it as anatomy, not as a tool for production access.

The TCP connection and port 23

Place Telnet above TCP, identify the well-known server port, and separate a successful connection from a working application session.

Telnet does not move bytes across the network itself. It sits on top of TCP and lets TCP do the transport work: delivery, ordering, retransmission. What Telnet adds is meaning, terminal semantics layered over a reliable byte pipe.

A Telnet session uses one full-duplex TCP connection. Both sides can send data and Telnet control information through the same byte stream, in both directions at once. There is no second connection for control traffic, which has consequences we will explore in a later lesson.

A Telnet server traditionally listens on TCP port 23. The client uses a temporary local port, chosen by the operating system, so the complete connection is identified by both addresses and both ports:

```
client 192.0.2.10:53144 -> server 192.0.2.20:23
```

That four-part identifier is why the server can hold sessions with many clients at the same time, and even several sessions from the same client machine. Each one differs in at least the client port.

You can see a live connection's two endpoints from the client side:

```
ss -tn | grep :23
# ESTAB  0  0  192.0.2.10:53144  192.0.2.20:23
```

Port 23 is a convention, not part of the protocol grammar. A Telnet service can listen elsewhere, and a Telnet client can connect to another TCP port. That second half is the interesting one: it is what turns the client into a general tool for poking at text protocols, as you will do later in this course.

Connected is not working

A completed TCP handshake proves that something accepted the connection. It does not prove that the service speaks Telnet correctly, that authentication will work, or that the session is safe.

This distinction sounds obvious and gets forgotten constantly. `Connected to 192.0.2.20` means exactly one thing: a process on that machine accepted a TCP connection on that port. It could be a Telnet server, an HTTP server someone started on the wrong port, or a firewall's decoy.

Keep the layers separate when reasoning about failures. ”Can I reach the port?” is a TCP question. ”Does the service behave?” is an application question. The whole troubleshooting module of this course builds on telling those apart.

The Network Virtual Terminal

Use the Network Virtual Terminal model to understand how different local terminals exchange characters and control functions.

Telnet had a hard problem to solve. In the 1970s, every vendor's terminal behaved differently. Different character sets, different line endings, different control keys. Connecting any terminal to any host would have required a translator for every pair of devices.

Telnet's answer is an imaginary terminal called the **Network Virtual Terminal**, or NVT. Neither side has to understand the other's hardware. Each side translates between its local terminal behavior and this common network representation.

The client does not need to know every detail of the server terminal, and the server does not need to know every client device. Each machine only needs two translations: local-to-NVT on the way out, NVT-to-local on the way in. This is the same trick as an intermediate representation in a compiler, or a common data format between services.

What the base NVT looks like

The base NVT uses seven-bit US-ASCII values inside eight-bit bytes. A carriage return followed by line feed represents a new line. A carriage return that should not move to a new line is followed by a zero byte:

```
new line:        CR LF     (bytes 13 10)
carriage return: CR NUL    (bytes 13 0)
```

The CR NUL form looks odd today, but it existed for printers, where "return to column one without advancing the paper" was a real operation. The rule that matters: on an NVT connection, a bare CR is never sent alone. It is always followed by LF or NUL.

You have already seen this in practice. When the lab server logged `68656c6c6f0d0a` for the word `hello`, the `0d0a` at the end was the NVT new line: CR LF.

The NVT also defines a small set of control functions, like interrupting a process or erasing a character. Those travel as IAC commands, which the next module covers in detail.

The starting point, not the ceiling

Later options can agree on richer behavior: eight-bit data, terminal types, window sizes. The simple NVT is the starting point when a connection opens, before either side assumes those options.

This default matters for robustness. Two implementations that share nothing else can still talk, because both are required to speak plain NVT until negotiation says otherwise.

Data and commands share one stream

Distinguish ordinary terminal data from Telnet control sequences when both travel through the same TCP byte stream.

Telnet puts terminal data and protocol commands in the same TCP stream. There is no separate control connection. Compare this with FTP, which opens one connection for commands and another for file data. Telnet chose one stream for everything, so it needs a way to mark where data stops and a command begins.

The marker is the byte value 255. It means **Interpret As Command**, abbreviated IAC. The next byte tells the receiver which Telnet command follows. Option negotiation commands include one more byte naming the option.

Here is what a receiver might pull out of the stream:

```
104 101 108 108 111        five data bytes: "hello"
255 253 1                  IAC DO ECHO (a three-byte negotiation)
255 246                    IAC AYT (a two-byte command)
119 111 114 108 100        five more data bytes: "world"
```

Every byte below 255 that is not part of a command sequence is plain terminal data. The receiver reads along, and the moment it hits 255 it switches to command parsing.

Escaping byte 255

What happens if terminal data genuinely contains byte 255? The sender writes it twice. The receiver interprets `255 255` as one data byte instead of the start of a command:

```
sender wants to transmit:  200 255 13
```

```
sender actually writes:    200 255 255 13
```

This is the classic escaping trick. You see the same idea in \\ inside strings and %% in format strings. Any protocol that mixes data and control in one channel needs one.

TCP does not respect your boundaries

TCP does not preserve application message boundaries. One read may contain half a command, several commands, or commands mixed with data. A read can even end exactly after an IAC byte, with the command code arriving in the next read.

A Telnet implementation must keep parsing until every complete sequence is available. In practice that means buffering: if the stream ends mid-sequence, hold the partial bytes and wait for more.

A naive parser that assumes "one read equals one message" works fine on a fast loopback connection and then breaks on a real network. That is exactly the kind of bug this lesson exists to spare you.

Check your understanding: Telnet foundations

Check Telnet roles, TCP, port 23, the Network Virtual Terminal, line endings, IAC, and byte-stream parsing.

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

Take this interactive quiz online

Commands and option negotiation

Read IAC commands, WILL and DO exchanges, echo behavior, and terminal subnegotiation.

Read IAC commands

Decode Telnet control sequences that interrupt, erase, synchronize, or negotiate behavior inside the shared byte stream.

Every Telnet command begins with IAC, byte 255. A simple command uses two bytes: IAC followed by its command code.

For example, IAC IP means Interrupt Process. IAC AYT means Are You There. IAC EC means Erase Character. These map to things a person at a terminal wants: stop the running program, check whether the far end is alive, undo the last typed character.

```
255 244  -> IAC IP   (Interrupt Process)
255 245  -> IAC AO   (Abort Output)
255 246  -> IAC AYT  (Are You There)
255 247  -> IAC EC   (Erase Character)
255 248  -> IAC EL   (Erase Line)
```

Why send `IAC EC` instead of a plain backspace character? Because these are protocol-level representations, so each server decides how they affect its local process. One system erases characters with backspace, another with delete. The NVT command is neutral, and each end translates it into whatever its local system means by "erase one character".

`IAC AYT` is worth trying by hand. Many Telnet clients let you send it from command mode:

```
telnet> send ayt
```

A responding server prints something visible, historically a message like `[Yes]`, to prove the connection is alive. It is a tiny liveness check built into the protocol.

The command codes 251 through 254 are WILL, WONT, DO, and DONT. Those are three bytes rather than two, because they carry an option code. They get the next lesson to themselves.

Parse the stream, not the packets

Do not search each TCP packet independently for these pairs. A packet may end after IAC while the command byte arrives in the next packet. From TCP's point of view that is completely legal: it promised you a byte stream, not tidy packages.

Parse the continuous stream. A correct reader consumes bytes in order, and when it sees 255 it waits for the next byte before deciding anything, even if that byte has not arrived yet.

The failure mode of getting this wrong is nasty: a parser that misses a split IAC treats the command byte as data. Byte 244 is not printable ASCII, so garbage lands in the session, and the intended interrupt never happens. Everything looks fine until packet boundaries fall in the wrong place.

Negotiate with WILL and DO

Read WILL, WONT, DO, and DONT as two independent conversations about who performs each Telnet option.

Telnet was built to connect wildly different systems. A 1970s mainframe and a modern laptop cannot assume the same terminal features, so options are negotiated instead of assumed. Both sides start from the plain NVT and upgrade from there, one option at a time.

Four commands express offers, requests, refusals, and instructions to stop:

- `WILL option`: I offer to perform this option
- `WONT option`: I refuse or will stop performing it
- `DO option`: please perform this option
- `DONT option`: do not perform, or stop performing, it

On the wire, each is a three-byte sequence. IAC, then the command, then the option code:

```
255 251 opt   IAC WILL opt
255 252 opt   IAC WONT opt
255 253 opt   IAC DO   opt
255 254 opt   IAC DONT opt
```

A side that agrees to `WILL ECHO` answers `DO ECHO`. A side that cannot support the offer answers `DONT ECHO`. The same works in reverse: you can request with `DO`, and the other side accepts with `WILL` or declines with `WONT`.

Refusal is always allowed. The rule from RFC 854 is that an option is only active once both sides have agreed. One side asking is never enough. This is why any implementation must handle a ”no” gracefully: the plain NVT keeps working either way.

Two independent conversations

Each direction is negotiated independently. Supporting an option while sending does not automatically mean the same option is active while receiving.

Take ECHO. `WILL ECHO` from the server means "I will echo what you send me". Whether the client also echoes in the other direction is a completely separate negotiation. Think of every option as two switches, one per direction, each needing agreement from both sides.

Avoid negotiation loops

Implementations must avoid negotiation loops. The classic bug: side A sends `DO ECHO`, side B acknowledges with `WILL ECHO`, side A treats the acknowledgment as a fresh offer and acknowledges back, and the two ping-pong forever.

The fix is to track the current state of each option. Do not acknowledge a state that is already active by sending the same request again and again. Only send a negotiation command when you want the state to change.

Echo and Suppress Go Ahead

Understand why Telnet negotiates who echoes typed characters and whether the old half-duplex Go Ahead signal is needed.

When you type a character into a terminal session, something has to display it. That job is called echo, and there are two ways to assign it.

With **local echo**, the client displays a character as you type it. With **remote echo**, the server sends the character back and the client displays the returned copy.

Telnet negotiates the ECHO option, defined in RFC 857 as option code 1, so both sides do not echo the same input. A typical remote-login server offers to do the echoing:

```
server: 255 251 1    IAC WILL ECHO  (I will echo what you type)
client: 255 253 1    IAC DO ECHO    (agreed, you echo)
```

After this exchange the client stops echoing locally and displays only what the server returns.

Get this wrong and you see it immediately. Two echoes duplicate characters, so typing `ls` shows `llss`. No echo makes typing appear invisible even though bytes may still be sent. Remote echo is also why old-school servers could hide your password: they agreed to echo, then chose not to echo those characters.

Suppress Go Ahead

The original NVT model assumed half-duplex terminals that could not send and receive at the same time. It included a Go Ahead signal, `IAC GA`, meaning "your turn to transmit".

Modern connections are full duplex, so the signal is pure overhead. Sessions normally negotiate **Suppress Go Ahead**, or SGA, option code 3, in both directions:

```
server: 255 251 3    IAC WILL SGA
client: 255 253 3    IAC DO SGA
```

Both directions matter because, as you learned in the previous lesson, each direction is a separate agreement. ECHO and SGA together are what turn a Telnet session into the character-at-a-time mode most people expect.

Echo is not buffering

Echo behavior and line buffering are separate ideas. A client may show characters locally while waiting for Enter before sending a line, or it may send each character immediately. One question is who paints the character on screen. The other is when bytes leave the machine. Keep them apart when you debug a strange-feeling session.

Subnegotiation and terminal details

Follow option-specific data between IAC SB and IAC SE, including terminal type and window dimensions.

WILL and DO can only say yes or no. That is enough for a switch like ECHO, but some options need more information after WILL and DO agree that the option may be used. "What terminal type are you?" cannot be answered with a yes. Telnet carries that data inside **subnegotiation**.

A subnegotiation begins with IAC SB, includes the option code and its data, then ends with IAC SE:

```
IAC SB option ...data... IAC SE
255 250 option ...data... 255 240
```

SB is byte 250 and SE is byte 240. Everything between them belongs to the named option and follows that option's own rules, defined in its own RFC.

NAWS: window size

The NAWS option, code 31, lets a client report its window width and height:

```
IAC SB NAWS 0 80 0 24 IAC SE
```

This example reports an 80-column, 24-row window. Each dimension uses two bytes in network byte order, so 80 is `0 80` and a width of 300 would be `1 44` (256 + 44). The server uses this to format output, and a good client sends a fresh NAWS subnegotiation whenever you resize the window.

TERMINAL-TYPE: a two-step dialogue

The TERMINAL-TYPE option, code 24, lets a server request a terminal name. After the client agrees with `WILL TERMINAL-TYPE`, the server asks and the client answers:

```
server: IAC SB TERMINAL-TYPE 1 IAC SE            (1 = SEND)
client: IAC SB TERMINAL-TYPE 0 x t e r m IAC SE  (0 = IS)
```

The name travels as plain ASCII characters inside the subnegotiation. Notice the pattern: negotiation decided *whether* to exchange terminal types, and subnegotiation carries *what* the type is.

The escaping rule still applies

Any data byte equal to 255 must still be doubled, even inside a subnegotiation. Otherwise a receiver could mistake it for the IAC that starts IAC SE.

This matters for NAWS specifically. A window 255 columns wide has a dimension byte of 255, so the client must send it twice. A parser that forgets this rule ends the subnegotiation early and misreads everything after it. If you ever write a Telnet parser, this is the test case to write first.

Check your understanding: commands and options

Check IAC commands, WILL and DO negotiation, refusals, echo, Suppress Go Ahead, subnegotiation, terminal type, and window size.

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

Take this interactive quiz online

Build a local Telnet lab

Create a loopback server, connect a client, use command mode, and inspect the bytes exchanged.

Choose a Telnet client

Check which Telnet client is available and keep the learning lab separate from production remote access.

Start by checking whether a Telnet client is already installed:

```
telnet
```

If you see a `telnet>` prompt, you have a client. Type `quit` and press Enter to leave. If the shell answers `command not found`, you need to install one.

Some operating systems include one. Others make it an optional package. macOS stopped bundling a Telnet client, and many Linux distributions leave it out of the default install:

```
# macOS
brew install telnet

# Debian / Ubuntu
sudo apt install telnet
```

Use the package supplied by your operating system, or a maintained client you can verify. This is a diagnostic tool that touches your network traffic. Don't grab a random binary from the web.

Two rules for this course

First, connect only to loopback services you start yourself or systems you are authorized to inspect. Poking at other people's ports is at best rude and at worst illegal.

Second, do not send a password through a plaintext Telnet session. Not once, not ”just to check”. Every byte you type crosses the network readable.

Why a Telnet client at all

The client is useful beyond remote login because it lets you type into a TCP connection and read the response. That is the whole trick. You see the exact bytes a text protocol exchanges, with no library hiding them.

It is not the only tool for this. **Netcat** is often a better raw TCP tool, because it sends exactly what you type and nothing else:

```
nc 127.0.0.1 2323
```

A Telnet client may inject protocol negotiation bytes into the stream. Netcat never does. We will use that difference deliberately in a later lesson.

When the service requires TLS, neither tool helps on its own. **OpenSSL** is the right starting point there:

```
openssl s_client -connect example.com:443
```

It performs the TLS handshake, then gives you the same type-and-read session on the decrypted stream.

My advice is to have all three installed. Telnet for this course, netcat for raw TCP work, `openssl s_client` for anything encrypted.

Start a loopback server

Create a tiny local TCP server that shows exactly which bytes a Telnet client sends without exposing a network service.

To study what a Telnet client actually sends, you need a server you fully control. Not a real telnetd, just something that accepts a TCP connection and shows you every byte. A few lines of Node.js are enough.

Create `server.mjs`:

```
import net from 'node:net'

const server = net.createServer(socket => {
  socket.write('Welcome\r\n')

  socket.on('data', data => {
    console.log(data.toString('hex'))
    socket.write(`Received ${data.length} bytes\r\n`)
  })
})

server.listen(2323, '127.0.0.1', () => {
  console.log('Listening on 127.0.0.1:2323')
})
```

The hexadecimal logging is the whole point. Printing incoming data as text would hide the bytes we care about most: line endings and Telnet protocol commands, which are not printable characters. In hex, nothing hides.

Run it in one terminal:

```
node server.mjs
```

You should see `Listening on 127.0.0.1:2323`. The process stays in the foreground, waiting. Leave it running; the next lesson connects to it.

Two deliberate choices in that `listen` call. Binding to `127.0.0.1` keeps the lab on your computer. The server is unreachable from the network, so you can experiment freely without exposing an unauthenticated service to your LAN. And port 2323 is an unprivileged alternative to the well-known Telnet port 23, which would require root to bind on Unix-like systems.

If the server exits immediately with `EADDRINUSE`, something already occupies port 2323, probably a previous run of the same script you forgot about. Find it with `lsof -i :2323` and stop it.

One thing this server is not: a Telnet server. It never sends IAC negotiation and never parses commands. It accepts TCP bytes and echoes statistics back. That neutrality is useful, because whatever appears in the hex log came from the client, not from us.

Use netcat as a tiny local server in one terminal:

```
nc -l 127.0.0.1 2323
```

Connect from another terminal:

```
telnet 127.0.0.1 2323
```

Type on both sides. You are seeing a TCP byte stream, not a full remote-login service. Stop both programs, then repeat with tcpdump on the loopback interface. This separates the TCP connection from Telnet option negotiation.

Connect and use command mode

Open the local connection, send terminal data, and leave the session through the client command escape.

Keep the server from the previous lesson running. Open another terminal and connect:

```
telnet 127.0.0.1 2323
```

The client prints a few lines before you type anything:

```
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Welcome
```

`Trying` means a TCP connection attempt is in progress. `Connected` means the handshake completed. `Welcome` is the first data your server sent. The escape character line matters most, and we will come back to it in a moment.

Type `hello` and press Enter. The server terminal prints the received bytes in hexadecimal:

```
68656c6c6f0d0a
```

That is `hello` in ASCII, followed by `0d0a`: carriage return and line feed. The client displays the server reply, `Received 7 bytes`. You typed five characters and the line ending added two more.

Command mode

A Telnet client also has a local command mode. The default escape is commonly `Ctrl+]`, which is what `'^]'` meant in the connection banner. Press it and the client shows its own prompt:

```
telnet>
```

You are now talking to the client program, not the server. Enter `status` to inspect the connection. It reports the connected host and the current escape character. Press Enter on an empty line to return to the session, or enter `quit` to close the connection and exit.

The escape character is handled by the client. It is not sent to the server as terminal data. Nothing appears in your server's hexadecimal log when you press it.

This gives you three layers to keep apart. Client commands like `status` and `quit` live in the client program only. Telnet protocol commands travel inside the stream as IAC sequences. Ordinary terminal data is everything else you type.

One common confusion: if you close the server while the client sits in command mode, the client may not notice until you return to the session and press a key. TCP only reports a closed connection when you try to use it.

Observe option negotiation bytes

Make the local server request a Telnet option and decode the client response as protocol bytes instead of printable text.

So far the lab server has only received data you typed. Now make it speak the Telnet protocol itself. Add this line immediately after the server accepts a connection:

```
socket.write(Buffer.from([255, 253, 31]))
```

The bytes mean IAC DO NAWS: the server asks the client to report window size. Byte 255 is IAC, 253 is DO, and 31 is the option code for **Negotiate About Window Size**, defined in RFC 1073.

A client that supports the option can answer IAC WILL NAWS and send a NAWS subnegotiation carrying its dimensions.

Restart the server, reconnect with the Telnet client, then inspect the hexadecimal log. You should see something like:

```
fffb1f
fffa1f00500018fff0
```

Decode it byte pair by byte pair. A response beginning `fffb1f` is IAC WILL NAWS: `ff` is 255, `fb` is 251 (WILL), `1f` is 31 (NAWS). The client accepted.

A later sequence beginning `fffa1f` is the NAWS subnegotiation. `fa` is 250, the SB command. The four data bytes that follow are the width and height, two bytes each. In this example `0050` is 80 columns and `0018` is 24 rows. The `fff0` at the end is IAC SE, closing the subnegotiation.

Resize your terminal window while connected. A well-behaved client sends a fresh `fffa1f` sequence with the new dimensions. You are watching live option traffic in your own log.

When the client says no

Client behavior varies. A refusal beginning `fffc1f` is also a valid result: IAC WONT NAWS. Netcat, for example, will not answer at all, because it is not a Telnet client and treats your three bytes as ordinary data.

Record what happened instead of assuming every client implements the same options. That habit is the whole point of this lab: negotiation is a conversation, and you only know its outcome by reading the actual bytes.

One thing to notice in your log: none of these bytes appeared on the client's screen. The client consumed them as protocol traffic. Data and commands share the stream, and this is what that looks like in practice.

Check your understanding: the local lab

Check safe lab boundaries, loopback binding, client command mode, hexadecimal traces, NAWS requests, and valid negotiation responses.

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

Take this interactive quiz online

Inspect text protocols

Use the Telnet client to test TCP and manually read HTTP and SMTP conversations.

Separate the tool from the protocol

Use a Telnet client as a manual TCP client without confusing the connected service with the Telnet protocol.

The `telnet` command can open a TCP connection to a host and port. This makes it useful for looking at line-oriented protocols, and it is why the tool outlived its original job.

If you connect to an HTTP server and type an HTTP request, the connection carries HTTP application data. The program is still a Telnet client, but the server is not a Telnet server:

```
telnet 127.0.0.1 8000
```

Nothing about this connection is "the Telnet protocol" except possibly some negotiation bytes the client sends. The server ignores or chokes on those, and the rest is pure HTTP. Saying "I telnetted to the web server" describes the tool you used, not the protocol you spoke.

The distinction sounds pedantic until it bites. Two things make a Telnet client an imperfect raw TCP tool.

First, this works best when the other protocol uses readable text and does not require TLS immediately. Against an HTTPS port the session dies at the handshake, because you type text where the server expects TLS records. `openssl s_client` is the tool for that case.

Second, Telnet command handling can also interfere if the data contains byte 255. The client is obligated to treat 255 as IAC, so binary data passing through gets reinterpreted or mangled. Netcat has neither problem:

```
nc 127.0.0.1 8000
```

It writes exactly what you type and prints exactly what arrives. When you want a guaranteed-transparent pipe, prefer it.

Know when to stop typing

Use the tool for learning and small authorized checks. Confirming a port answers, reading a banner, sending one test command: perfect fits.

Use a protocol-specific client for real work because it understands framing, validation, authentication, redirects, timeouts, and errors. Typing HTTP by hand teaches you what `curl` does. It does not replace `curl`, which handles chunked encoding, retries, and TLS without you thinking about any of it.

The mental model to keep: TCP carries bytes, a protocol gives the bytes meaning, and a client program is just how the bytes get typed. Any tool that can push bytes into a socket can speak any text protocol, badly or well.

Test TCP reachability

Interpret a successful or failed connection as narrow evidence about DNS, routing, firewalls, listening ports, and the application.

"Is that port even reachable?" is one of the most common questions in debugging, and a Telnet client answers it in seconds. You can ask it to connect to a specific TCP port:

```
telnet 127.0.0.1 8000
```

There are three common outcomes, and each one means something different:

```
Connected to 127.0.0.1.
telnet: Unable to connect to remote host: Connection refused
telnet: Unable to connect to remote host: Operation timed out
```

What success proves

A successful connection proves that the client resolved the address, reached the host, completed a TCP handshake, and found something listening on that path. That is four useful facts, and if you were debugging "the app can't reach the database", they eliminate DNS, routing, and firewall in one test.

It does not prove that the service is healthy. The application may speak the wrong protocol, reject the next command, return bad data, or wait forever. Something answered the handshake; you know nothing yet about what.

What failure tells you

A refusal usually means the host is reachable but nothing accepted that port. The machine actively answered "no". Typical causes: the service is not running, it crashed, or it listens on a different port or address. Check the server side with `ss -tlnp` and compare what is actually bound against what you assumed.

A timeout may point to routing, firewall, address, or silent packet loss. Your packets got no answer at all, and you cannot tell from the client alone whether they never arrived or the replies were dropped. Firewalls that silently discard packets produce exactly this symptom, and it takes noticeably longer to appear than a refusal.

A subtle trap: a refusal from `127.0.0.1` when the service is up sometimes means it bound only to another address. A server listening on `192.168.1.5:8000` refuses loopback connections. The port is fine, the address was wrong.

Preserve the exact result before changing configuration. "Refused" versus "timed out" sends you down entirely different paths, and if you restart three things before reading the message carefully, you have lost the evidence.

Send an HTTP request by hand

Type a complete HTTP/1.1 request through a Telnet client and see the protocol message carried inside TCP.

Create `http-server.mjs` so the exercise has a local HTTP server:

```
import http from 'node:http'

http.createServer((request, response) => {
  response.end('Hello from HTTP\n')
}).listen(8000, '127.0.0.1')
```

Run it with `node http-server.mjs`. Then connect its port with the Telnet client:

```
telnet 127.0.0.1 8000
```

Type these lines, then press Enter once more to send the empty line that ends the headers:

```
GET / HTTP/1.1
Host: localhost
Connection: close
```

The server responds with a status line, headers, an empty line, and usually a body. `Connection: close` makes the end easy to observe because the server closes the TCP connection after the response.

You just used a Telnet client to speak HTTP. The request grammar came from HTTP, while TCP transported the bytes and the Telnet program provided the interactive connection.

Connect to a local HTTP server or a host you control, then type a complete request followed by a blank line:

```
GET / HTTP/1.1
Host: localhost
Connection: close
```

Read the status line, headers, blank line, and body. Telnet does not understand HTTP here. It only carries the bytes you type. Repeat with curl -v and compare the exact request and response boundaries.

Read an SMTP greeting

Inspect a plaintext SMTP greeting and EHLO reply without authenticating, relaying mail, or exposing credentials.

SMTP is another line-oriented text protocol, and unlike HTTP, the server speaks first. In an authorized mail lab, connect to the SMTP server port:

```
telnet mail.lab.test 25
```

A server normally starts with a 220 greeting before you type anything:

```
220 mail.lab.test ESMTP Postfix
```

Send an EHLO name, read the advertised extensions, then quit:

```
EHLO client.lab.test
QUIT
```

A real exchange looks like this:

```
220 mail.lab.test ESMTP Postfix
EHLO client.lab.test
```

```
250-mail.lab.test
250-PIPELINING
250-SIZE 10240000
250-STARTTLS
250 SMTP UTF8
QUIT
221 2.0.0 Bye
```

Every server reply starts with a three-digit code. `220` is the greeting, `250` means success, `221` says goodbye. Your client software can branch on the code without parsing the human-readable text after it.

Reading the multiline reply

Look closely at the EHLO response. A multiline EHLO reply uses `250-` on every line except the last, which uses `250 ` with a space. The hyphen means "more lines follow with this same code". The space means "this is the final line".

This small detail demonstrates how a text protocol defines message boundaries inside a TCP stream. TCP will not tell the client where the reply ends. The protocol grammar does, one character at a time. A client that stops reading after the first `250-` line has a framing bug, and it is exactly the kind of bug you catch by reading the raw exchange like this.

The extension list is the useful diagnostic output here. `STARTTLS` tells you the server can upgrade to an encrypted connection. `SIZE` tells you the maximum message it accepts.

Where to stop

Stop before AUTH and do not send a password. Everything you typed in this session crossed the network in plaintext, and credentials must never travel that way.

Modern submission on port 587 commonly requires TLS before authentication, so a plaintext session cannot get far anyway. Use a TLS-aware mail client or `openssl s_client -starttls smtp -connect mail.lab.test:587` when you need to inspect that path. You get the same readable dialogue, after encryption is in place.

Check your understanding: inspecting protocols

Check the difference between tool and protocol, TCP reachability evidence, HTTP message boundaries, SMTP replies, and TLS limits.

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

Take this interactive quiz online

Security and troubleshooting

Recognize plaintext risks, replace remote login with SSH, diagnose failures, and design a tiny protocol.

Understand the plaintext risk

Identify what an observer or active attacker can learn and change in an ordinary Telnet remote-login session.

Classic Telnet does not provide confidentiality, server authentication, or integrity protection. Every byte crosses the network exactly as typed. This is the single most important fact in this course, and it is why Telnet remote login is dead.

A network observer can read terminal output and input, including usernames, passwords, commands, and returned data. Anyone positioned on the path sees your whole session as readable text: a compromised router, a rogue Wi-Fi access point, another tenant on a shared network segment.

You can prove this to yourself with the loopback lab. Capture the traffic while a session is running:

```
sudo tcpdump -i lo0 -A port 2323
```

Use `-i lo` on Linux. The `-A` flag prints packet contents as ASCII. Now type into your Telnet session and watch the capture:

```
12:04:31.118 IP 127.0.0.1.53144 > 127.0.0.1.2323: Flags [P.]
E..5..@.@.............  .hello
```

There is `hello`, readable in the packet dump. If that had been a password, the observer would have it. No decryption step, no cracking, just reading.

Active attackers do worse

An active attacker may alter the byte stream or impersonate the server. Without integrity protection, nothing detects a modified command in flight. Without server authentication, nothing proves the login prompt you see belongs to the machine you meant to reach. A fake server collects your password and forwards your session onward, and you never notice.

What negotiation does not fix

Option negotiation does not add security. Agreeing on echo, terminal type, or window size changes terminal behavior, not trust or encryption. A fully negotiated, perfectly behaving Telnet session is exactly as exposed as a bare one.

The same goes for network position. Never use ordinary Telnet remote login across an untrusted network. A private address or unusual port does not encrypt the connection. Moving telnetd to port 9923 on `10.0.0.5` is obscurity, and any observer inside that network still reads everything.

The only session in this course you should ever type real input into is one where plaintext is the point: your own loopback lab, carrying nothing secret. The next lesson covers the actual replacement.

Replace remote login with SSH

Choose SSH for real remote administration and understand which security properties it adds before user commands run.

The previous lesson showed what an observer gets from a plaintext session: everything. The fix is not a Telnet patch or a hidden port. Use SSH for remote login over an untrusted network. This is not a preference, it is the baseline. Telnet remote login has been obsolete for this job since the 1990s.

SSH establishes a protected transport, authenticates the server, authenticates the user, and carries terminal sessions through encrypted channels. It also protects the integrity of the traffic, so an attacker cannot silently alter bytes in flight. Every property Telnet lacks, addressed before your first command runs.

```
ssh alice@server.example
```

On the first connection, SSH shows you the server's host key fingerprint:

```
The authenticity of host 'server.example (203.0.113.7)' can't be established.
ED25519 key fingerprint is SHA256:xk8m2P1vQ9rT4wLnYc6bJdHs0aZeUf3gRi7oN5qWvKk.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Do not type `yes` on reflex. Host-key verification is essential. This prompt is SSH asking you to confirm you are talking to the right machine, ideally by comparing the fingerprint against one you obtained through a trusted channel. An encrypted connection to an unverified attacker is not a safe replacement. Encryption without authentication just means the attacker reads your session privately.

Once accepted, the key is stored in `~/.ssh/known_hosts` and checked on every later connection. A sudden mismatch warning means the server changed or someone is intercepting you. Stop and find out which.

Check nothing still listens on 23

While replacing Telnet, verify the old service is actually gone on your servers:

```
ss -tln | grep :23
```

No output is the answer you want. A telnetd that keeps running next to sshd keeps the plaintext risk alive for anyone who still connects to it.

What Telnet is still for

Keep Telnet for controlled legacy systems, protocol study, and narrow diagnostics where plaintext is intentional and no secret is sent. That is exactly how this course uses it: a window into TCP and text protocols. For logging into machines and running commands, the answer is SSH, every time.

Diagnose a failed session by layer

Separate name resolution, TCP connectivity, Telnet negotiation, authentication, terminal behavior, and application failures.

Start with the exact symptom. ”Telnet does not work” hides several different failures, and each one lives at a different layer.

The client output usually tells you which layer failed, if you read it carefully:

```
telnet: could not resolve badname.lab.test: Name or service not known
telnet: Unable to connect to remote host: Connection refused
telnet: Unable to connect to remote host: Operation timed out
```

If the name does not resolve, inspect DNS. The connection never started, so firewalls and servers are not suspects yet. Test with `dig` or try the raw IP address instead.

If the connection is refused, the host answered but nothing listens on that port. Inspect the address, the port number, and the listening process on the server side:

```
ss -tlnp | grep 2323
```

If the connection times out, packets are disappearing. Inspect the route, the firewall, and the address. A refusal is an answer; a timeout is silence. They point at different problems.

After the connection succeeds

If bytes arrive but look strange, inspect Telnet negotiation and terminal settings. A connected session with duplicated or invisible input usually points to echo negotiation. Broken screen layout may point to terminal type or window size.

A login rejection belongs to authentication or account policy. The network did its job: your connection worked, your credentials or account did not. Do not restart the network stack for a wrong password.

An application that accepts the connection and then misbehaves is yet another layer. The service may speak a different protocol than you expected, or be unhealthy behind a working port.

Work methodically

Change one layer at a time. If you change the port, the firewall rule, and the server config in one step, a fixed session teaches you nothing about which change mattered.

Preserve the client message, server log, and a byte trace when negotiation is involved. Evidence is more useful than repeatedly changing ports or disabling checks. My advice is to write down each test and its result as you go. Five minutes of notes beats an hour of re-testing things you already ruled out.

Design a tiny text protocol

Use the local TCP lab to define commands, replies, framing, errors, and connection closure for a protocol of your own.

You have read HTTP and SMTP by hand. Now design a protocol of your own. Extend the local Node.js server with three commands: `TIME`, `ECHO text`, and `QUIT`.

Choose one line ending and buffer input until a complete line arrives:

```
let buffer = ''

socket.on('data', chunk => {
  buffer += chunk.toString('utf8')
  let index
  while ((index = buffer.indexOf('\r\n')) !== -1) {
    const line = buffer.slice(0, index)
    buffer = buffer.slice(index + 2)
    handleLine(socket, line)
  }
})
```

The `while` loop matters. One `data` event may carry zero, one, or three complete lines. TCP supplies a stream, so one `data` event is not the same as one command.

Return one numeric reply per command, SMTP-style:

```
200 2026-08-03T10:00:00Z
200 hello
400 unknown command
221 bye
```

Numeric codes let a client program check the first three characters without parsing the whole line. Humans read the text after the code. Machines read the code.

Connect with your Telnet client and test the happy path:

```
TIME
200 2026-08-03T10:00:00Z
ECHO hello
200 hello
QUIT
221 bye
Connection closed by foreign host.
```

Then test the ugly paths. Send an unknown command and confirm you get `400 unknown command` instead of silence. Test fragmented input and two commands arriving in one read. Netcat can do that: `printf 'TIME\r\nECHO hi\r\n' | nc 127.0.0.1 2323` delivers two commands in a single write.

Write the contract down

Document the command grammar, reply grammar, encoding, maximum line length, timeout, and close behavior. A protocol without a maximum line length invites a client to send an endless line and eat your memory. A protocol without a timeout collects dead connections forever.

You now have an application protocol that a Telnet client can demonstrate. It is not the Telnet protocol unless you also implement Telnet commands and negotiation. The distinction from earlier lessons applies to your own work too: the tool speaks TCP, your protocol defines what the bytes mean.

Check your understanding: security and troubleshooting

Check plaintext exposure, SSH properties, layered diagnosis, echo failures, TCP stream framing, and the final protocol design.

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/telnet/.