PROGRAMMING

API Testing & Tooling

Postman Basics, Collections & Scripting

VS Code as a REST Client · cURL Fundamentals & Advanced

Browser-Based Testing · Capstone: All Four Tools

Philip Osztromok

Generated with Claude

Table of Contents

API Testing & Tooling — 9 Chapters

🧪 Why API Testing Matters & The Tools Landscape

The **API Types & Design** course covered how APIs are built — REST, SOAP, RPC-style, webhooks. This course is the hands-on companion: actually poking at a real API with real tools, reading what comes back, and building the habit of testing before assuming something works. Four genuinely different tool categories are covered, each suited to a different situation.

Manual vs. Automated Testing

Manual testing means a person builds and fires off a request by hand, right now, to answer a specific question — "does this endpoint return what I expect?" It's fast to start, needs no setup, and is exactly right for one-off exploration or debugging something in front of you. **Automated testing** means the same checks are captured as code or a saved script, so they can be re-run identically — by a person, on a schedule, or as part of a CI pipeline — without a human repeating the same manual steps every time.

Manual vs. Automated, Side by Side

Manual

A developer opens Postman, fills in a URL, clicks Send, reads the response. Perfect for "let me just check this real quick" — but nothing stops the exact same bug from resurfacing next week unnoticed.

Automated

A saved test script runs the same request and asserts on the response every time — in a CI pipeline, on every commit. Slower to set up once, but the check never gets forgotten or skipped.

Neither replaces the other. This course builds manual skill first (Postman, `.http` files, cURL, the browser) because automated testing is really just "the same manual check, captured and repeated"

— Chapter 4's Newman CLI runner and Chapter 9's capstone both build directly on manual skills from earlier chapters.

The Four Tool Categories This Course Covers

Category	What It's Best At	Covered In
Postman	Building and organizing requests visually; the most common starting point	Chapters 2–4
VS Code as a REST Client	Version-controllable, plain-text request files living right next to your code	Chapter 5
cURL	Scriptable, terminal-native, works anywhere a shell does — no GUI needed	Chapters 6–7
Browser-Based Testing	Zero setup — testing a GET endpoint with nothing but the address bar or DevTools	Chapter 8

When to Reach for Which

Reach for Postman When

You're exploring an unfamiliar API, need to save and organize many related requests, or want to chain requests together (Chapter 3) without writing a full script.

Reach for a VS Code .http File When

You want the request definitions themselves committed to the same Git repository as the code that calls them — reviewable in a pull request, not locked inside a GUI app's local state.

Reach for cURL When

You're already in a terminal (SSH session, a shell script, a CI job) and there's no GUI available at all — or you want a request that's trivially copy-pasteable into a bug report.

Reach for the Browser When

It's a simple GET request and you want the fastest possible check — paste a URL, look at what comes back, no tool to open at all.

The Same Request, Previewed Across All Four

Checking a public endpoint that returns a joke — this exact request reappears, tool by tool, across Chapters 2–8:

```
// Postman: a saved GET request in a collection (Ch.2–3)
GET https://api.example.com/joke

// VS Code .http file (Ch.5)
GET https://api.example.com/joke

// cURL (Ch.6)
curl https://api.example.com/joke

// Browser address bar (Ch.8)
https://api.example.com/joke
```

Same request, four different surfaces — the point of this course is knowing which surface fits the moment, not treating one tool as universally "the" way to test an API.

💻 Coding Challenges

Challenge 1: Manual or Automated?

Classify each as manual or automated testing, and justify: (a) a developer pasting a URL into a browser to check a fix just now, (b) a GitHub Actions workflow hitting an endpoint on every pull request, (c) clicking "Send" in Postman while debugging live.

Goal: Practice the manual-vs-automated distinction on realistic, not purely textbook, examples.

→ Solution

Challenge 2: Pick the Right Tool

For each scenario, name the best-fit tool category from this chapter and justify it: (a) quickly checking whether a public GET endpoint is up, (b) a teammate needs to reproduce your exact request from a bug report with no Postman installed, (c) a request definition that should be reviewed in a pull request alongside the code that calls it.

Goal: Practice applying the "when to reach for which" decision guide to concrete situations, not just memorizing the table.

→ Solution

Challenge 3: Why Not Just One Tool?

A developer says: "Postman does everything I need — why bother learning cURL or .http files at all?" Write a short response explaining at least two genuinely different situations where Postman isn't the best fit.

Goal: Practice articulating the trade-offs behind this course's four-tool structure, rather than assuming more features always means a better default choice.

→ Solution

⚠️ Gotcha: Testing Only the Happy Path

It's tempting to test an endpoint exactly once, with exactly the input you expect it to receive, see a 200 response, and call it "tested." Real usage is never that clean — a missing field, a malformed value, an expired token, a request for something that doesn't exist. A genuinely useful test checks the **error** responses too: does a bad request actually return a 400 with a useful message, or does it 500 and leak a stack trace? This course's later chapters (especially the test-scripting in Chapter 4 and the capstone in Chapter 9) build this habit in directly — get comfortable, starting now, with deliberately sending a request you expect to *fail*, not just ones you expect to succeed.

🎯 What's Next

The next chapter gets hands-on with the most common starting point: **Postman Basics** — building a request, setting headers/params/body, and reading what comes back.

Postman Basics

Chapter 1 named Postman as the most common starting point for API testing. This chapter builds the first real request from scratch — method, URL, params, headers, body — and reads what comes back.

The Postman Interface, in Brief

A request builder in Postman has, left to right: a **method dropdown** (GET, POST, PUT, DELETE…), a **URL bar**, and a row of tabs — **Params**, **Headers**, **Body**, and a few others this chapter doesn't need yet. Below that, after clicking **Send**, a response pane shows what came back.

Building Your First Request

The simplest possible request: pick `GET`, type a URL, click Send.

```
GET https://api.example.com/products/42
```

The response pane immediately shows three things worth reading before even looking at the body: the **status code** (e.g. `200 OK`), the **response time** (e.g. `184 ms`), and the **response size**. All three are useful signal on their own — a `404` means "look at the URL, not the body," a jump from 100ms to 3 seconds is worth investigating regardless of whether the body looks correct.

Query Parameters

The **Params** tab lets you add key/value pairs without hand-typing a `?key=value&key2=value2` string — Postman builds the URL for you as you type, and keeps each parameter individually toggleable (useful for quickly testing "what if I remove just this one filter").

```
// Params tab: category = electronics, limit = 10
// Postman builds this URL automatically:
GET https://api.example.com/products?category=electronics&limit=10
```

Headers

The **Headers** tab adds custom HTTP headers — most commonly an `Authorization` header carrying a token, or a `Content-Type` header describing the body's format.

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
Content-Type: application/json
```

Request Body

For a `POST` or `PUT`, the **Body** tab is where the actual payload goes. Selecting **raw** and then **JSON** as the format gives a text area for hand-writing a JSON object, with Postman validating the syntax as you type.

```
POST https://api.example.com/products

// Body (raw, JSON):
{
  "name": "Wireless Mouse",
  "price": 24.99,
  "category": "electronics"
}
```

Inspecting the Response

The response side has its own tabs mirroring the request's: **Body** (with Pretty/Raw/Preview views — Pretty auto-formats JSON for readability), **Cookies**, and **Headers** (worth checking for things like rate-limit counters or a returned `Location` header after a successful creation).

Tab / Element	Purpose
Params	Query string key/value pairs, auto-appended to the URL
Headers	Custom HTTP headers sent with the request
Body	The request payload — raw JSON, form data, or other formats
Response status/time/size	The first three things worth checking before reading the body at all
Response Body (Pretty)	Auto-formatted, readable view of a JSON response

Query Params vs. Request Body — When Each Is Used

> #### Query Parameters
>
> Filtering, sorting, and pagination on a `GET` — small, URL-safe values that describe *which* data to fetch.

> #### Request Body
>
> The actual data being sent to create or update something on a `POST`/`PUT` — can be arbitrarily large and structured, and never appears in the URL.

Saving Requests

Clicking **Save** and giving a request a name keeps it in Postman rather than losing it the moment the tab closes. A saved request on its own is useful, but Chapter 3 makes it much more useful — organizing saved requests into **collections** and reusing values across them with **environments**.

Putting It Together: Creating a Product

Method `POST`, URL `https://api.example.com/products`, a `Content-Type: application/json` header, and a JSON body — then reading a `201 Created` response back:

```
// Response: 201 Created, 92ms
{
 "id": 108,
 "name": "Wireless Mouse",
 "price": 24.99,
 "category": "electronics"
}
```

The server echoed back the created product, now with an assigned `id` — confirming both that the request was understood and exactly what got stored.

💻 Coding Challenges

Challenge 1: Params or Body?

For each, say whether it belongs in query parameters or the request body, and why: (a) filtering a product list by `category`, (b) the full contents of a new blog post being created, (c) a page number for pagination.

Goal: Practice applying this chapter's params-vs-body distinction to concrete request-design decisions.

→ Solution

Challenge 2: Build a Request From a Description

Write out the method, URL with query parameters, headers, and body for: "create a new user with name 'Dana' and email 'dana@example.com', authenticating with a bearer token."

Goal: Practice assembling every piece of a request (method, URL, headers, body) from a plain-English description.

→ Solution

Challenge 3: Diagnose From Status Alone

Before even opening the response body, what does each of these status codes already tell you, and where should you look first: (a) 404, (b) 401, (c) 500?

Goal: Practice using the status code as a first diagnostic signal, per this chapter's "read status/time/size before the body" habit.

→ Solution

⚠️ Gotcha: A JSON Body With No Content-Type Header

Typing a JSON object into the Body tab's raw editor doesn't automatically tell the *server* it's JSON — that's what the `Content-Type: application/json` header is for. Postman sets this header automatically when you select "JSON" from the raw body's format dropdown, but if that dropdown gets left on "Text" (or the header gets manually deleted), the server may receive a perfectly well-formed JSON body with no indication of its format, and either fail to parse it or silently treat every field as plain text. If a request with a body you're sure is correct keeps failing, checking the actual `Content-Type` header being sent is one of the first things worth verifying.

🎯 What's Next

The next chapter is **Postman Collections & Environments** — organizing saved requests into collections, reusing values across requests with environment variables, and chaining one request's response into the next.

🗂️ Postman Collections & Environments

Chapter 2 ended by saving one request. That works for one request — it stops working the moment there are thirty of them, scattered across dev, staging, and production servers. This chapter fixes both problems: **collections** organize related requests, and **environments** let the same collection point at a different server without editing a single request.

Collections: Organizing Related Requests

A **collection** is a named group of saved requests — and folders inside a collection group them further. A "Shop API" collection might have a `Products` folder, an `Orders` folder, and a `Users` folder, each holding the requests relevant to that part of the API, instead of one long unsorted list that gets harder to navigate with every request saved.

```
Shop API (collection)
├── Products
│   ├── List Products
│   ├── Get Product by ID
│   └── Create Product
├── Orders
│   ├── Create Order
│   └── Get Order Status
└── Users
    ├── Login
    └── Get Current User
```

Environment Variables

Hardcoding `https://api.example.com` into thirty saved requests means editing thirty requests the day that URL needs to change — or worse, needing a completely different URL for local development versus production. An **environment** defines variables like `baseUrl`, and every request in the collection references `{{baseUrl}}` instead of a literal URL.

```
// The request itself, written once:
GET {{baseUrl}}/products/42

// Dev environment:     baseUrl = http://localhost:3000
// Staging environment:  baseUrl = https://staging.example.com
// Production environment: baseUrl = https://api.example.com
```

Switching the active environment in Postman's environment dropdown changes where *every* request in the collection points, without touching any of them individually.

Collection Variables vs. Environment Variables

Two Kinds of Variable, Two Different Jobs

Collection Variables

Shared across every environment — good for values that don't change between dev/staging/prod, like an API version path segment (`/v2`) used the same way everywhere.

Environment Variables

Specific to whichever environment is currently active — good for values that genuinely differ per environment, like the base URL itself or an environment-specific API key.

Scope	Typical Use
Global	Rarely used deliberately — visible everywhere, in every collection; easy to lose track of what set it
Collection	Shared across all environments for this one collection (e.g. a fixed API version path)
Environment	Differs by active environment (base URL, environment-specific credentials)
Local (a single request's own overrides)	Temporary, request-specific tweaks that shouldn't affect anything else

Chaining Requests: Using One Response in the Next

A common real pattern: logging in returns an auth token, and every subsequent request needs that token in its `Authorization` header. Without chaining, that means manually copying the token out of one response and pasting it into the next request, every single time. Postman's response pane has a quicker path: selecting a value in the response and choosing **"Set as variable"** saves it directly into the active environment — no copy-paste required, and no code, which is deliberately left for Chapter 4's scripting.

Chaining a Login Into an Authenticated Request

```
// Request 1: Login
POST {{baseUrl}}/login
Body: { "email": "dana@example.com", "password": "..." }

// Response: { "token": "eyJhbGciOi..." }
// -> Select the token value -> "Set as variable" -> saved as {{authToken}}
// Request 2: Get Current User, reusing that token
GET {{baseUrl}}/me
Authorization: Bearer {{authToken}}
```

Request 2 never hardcodes a token — it reads whatever `{{authToken}}` currently holds, which Request 1 just set. Re-running Request 1 automatically refreshes it for every request that follows.

💻 Coding Challenges

Challenge 1: Organize a Collection

A "Blog API" needs requests for: listing posts, getting one post, creating a post, listing comments on a post, and creating a comment. Propose a folder structure for this collection.

Goal: Practice grouping related requests logically, the same way this chapter's Shop API example was organized by resource.

→ Solution

Challenge 2: Collection or Environment Variable?

For each, say whether it should be a collection variable or an environment variable, and why: (a) `baseUrl`, (b) an API version path segment used identically in dev, staging, and production, (c) a per-environment API key.

Goal: Practice applying the "shared everywhere vs. differs per environment" test to concrete variables.

→ Solution

Challenge 3: Design a Chained Sequence

Describe the request chain (using this chapter's "Set as variable" pattern) needed to: create a new order, then immediately fetch that order's status using the ID the creation response returned.

Goal: Practice identifying which value needs to flow from one response into the next request's URL, not just its headers.

→ Solution

⚠️ Gotcha: Exporting an Environment With Real Secrets Still Baked In

Sharing or backing up a Postman environment often means exporting it as a JSON file — and that export includes every variable's **current value**, not just its name. An environment holding a real production API key or a live auth token, exported and then committed to a shared repository or sent over chat, leaks that secret exactly like the hardcoded-credential mistake covered elsewhere on this site — the fact that it's "just a Postman file" doesn't make the value inside it any less sensitive. Treat an exported environment file with real production values the same way you'd treat a `.env` file: never commit it, and prefer Postman's built-in "secret" variable type (which masks the value in the UI) for anything genuinely sensitive.

🎯 What's Next

The next chapter is **Postman Scripting & Automation** — pre-request scripts, test scripts with `pm.test`, running a whole collection unattended with the Newman CLI runner, and mock servers.

⚙️ Postman Scripting & Automation

Chapter 3's "Set as variable" click worked, but it's still a manual step someone has to remember every time. This chapter is where Postman crosses from Chapter 1's "manual" column into "automated" — scripts that run assertions and set variables by themselves, and a command-line runner that executes an entire collection with no Postman UI open at all.

Pre-Request Scripts

A **pre-request script** is JavaScript that runs *before* the request is sent — useful for computing something the request needs on the fly, like a current timestamp or a signature.

```
// Pre-request Script tab
pm.environment.set("requestTime", Date.now());
```

Test Scripts: pm.test()

A **test script** runs *after* the response arrives, and is where real assertions live — checking a status code, a header, or a specific field's value, using `pm.test()` blocks and Chai-style `pm.expect()` assertions.

```
pm.test("Status code is 200", () => {
  pm.response.to.have.status(200);
});

pm.test("Response has a product name", () => {
  const data = pm.response.json();
  pm.expect(data.name).to.be.a("string");
});
```

Every test in the script runs automatically each time the request runs — no one has to remember to manually check anything, and Postman's runner shows a clear pass/fail per test.

Chapter 3's Manual Step vs. This Chapter's Automated One

> **Manual (Chapter 3)**
>
> A person selects the token in the response body and clicks "Set as variable" — has to remember to do it, every single time the login request runs.

> **Automated (This Chapter)**
>
> A test script does the exact same thing in code, every time, with no one needing to remember anything: `pm.environment.set("authToken", pm.response.json().token);`

API	Purpose
`pm.response.to.have.status(code)`	Assert the response status code
`pm.response.json()`	Parse the response body as JSON
`pm.expect(value).to...`	Chai-style assertion on any value
`pm.environment.set(name, value)`	Store a value into the active environment
`pm.environment.get(name)`	Read a value from the active environment

Automating Chapter 3's Login Chain

The login request's Test script now does in one automatic step what Chapter 3 did by hand:

```
pm.test("Login succeeded", () => {
  pm.response.to.have.status(200);
});

const body = pm.response.json();
pm.environment.set("authToken", body.token);
```

Every subsequent request in the collection referencing `{{authToken}}` automatically gets whatever this run just extracted — no manual click required, and it now fails loudly (a red test) if login itself ever stops returning a `200`.

Newman: Running a Collection From the Command Line

Newman is Postman's official CLI runner — it executes an entire collection (with all its test scripts) with no Postman application open at all, printing a pass/fail summary and exiting with a non-zero code if anything failed.

```
npm install -g newman
newman run shop-api.postman_collection.json -e production.postman_environment.json
```

This is the concrete bridge from "a Postman collection I click through" to "an automated regression suite" — the same collection built by hand in Chapters 2–3 can now run unattended, in a CI pipeline, on every commit.

Mock Servers

Postman can generate a **mock server** from a collection's saved example responses — a fake but realistic backend that returns those saved examples for matching requests. This lets a frontend team build and test against realistic API responses before the real backend exists, or lets a test suite run without depending on a real backend that might be slow, rate-limited, or still under construction.

💻 Coding Challenges

Challenge 1: Write a Test Script

Write a test script for a `GET /products/42` request that asserts: the status is 200, and the response body's `id` field equals `42`.

Goal: Practice writing two independent `pm.test()` blocks using `pm.response.to.have.status()` and `pm.expect()`.

→ Solution

Challenge 2: Automate a Chained Variable

A "Create Order" request returns `{ "id": "0123", "status": "processing" }`. Write the test script line that automatically saves `id` into an environment variable called `orderId`, the automated version of Chapter 3's manual chaining challenge.

Goal: Practice converting a manual "Set as variable" step into a one-line automated equivalent.

→ Solution

Challenge 3: Explain Newman's Role in CI

Explain what running `newman run collection.json` as a step in a CI pipeline actually verifies, and what a non-zero exit code should cause the pipeline to do.

Goal: Practice connecting Newman's exit code to CI/CD Pipelines course concepts (a failing test should fail the build, per `pipelines1-4`).

→ Solution

⚠️ Gotcha: A Swallowed Newman Exit Code Means CI Reports Green on a Failing Test

Newman exits with a non-zero code when any test fails — that's exactly what a CI pipeline is supposed to detect and fail on. But it's an easy mistake to wrap the command in a way that discards that signal, e.g. `newman run collection.json || true`, or piping its output through another command that doesn't propagate the original exit code. The pipeline then reports success even though real API tests just failed — silently defeating the entire point of running Newman in CI in the first place. This is the exact same discipline `pipelines1-4` established for any test step: the build must fail when the tests do, with nothing in between quietly absorbing that failure.

🎯 What's Next

The next chapter steps outside Postman entirely: **VS Code as a REST Client** — the REST Client extension's version-controllable `.http` files, and Thunder Client as a Postman-like in-editor alternative.

📝 VS Code as a REST Client

Chapter 1 named "requests that need to live in version control, reviewable in a pull request" as a case Postman doesn't fit well. This chapter delivers on that directly: the REST Client extension turns API requests into plain-text files that sit in the same Git repository as the code that calls them.

Why a Plain-Text Request File?

A Postman collection lives inside Postman's own application state (or a separately exported JSON file). An `.http` file is just a text file — it sits in the repo, shows up in `git diff`, gets reviewed in a pull request like any other change, and needs no export/import step to stay in sync with the code around it.

The REST Client Extension

After installing the REST Client extension, any file with a `.http` (or `.rest`) extension gets a clickable **"Send Request"** link above each request, right in the editor. The syntax is close to raw HTTP: a method and URL, optional header lines directly below, a required **blank line**, then an optional body.

```
GET https://api.example.com/products/42

###

POST https://api.example.com/products
Content-Type: application/json

{
  "name": "Wireless Mouse",
  "price": 24.99
}
```

`###` on its own line separates multiple requests within one file — each gets its own "Send Request" link.

Variables in .http Files

Variables use `@name = value` at the top of the file, referenced later with `{{name}}` — the same mental model as Postman's environment variables (Chapter 3), just written directly as plain text instead of managed through a UI.

```
@baseUrl = https://api.example.com

GET {{baseUrl}}/products/42
```

For values that genuinely differ per environment (and especially anything sensitive), the extension also supports a separate `http-client.env.json` file defining named environments — the `.http` file itself stays identical across dev/staging/production, referencing whichever environment is currently selected.

Chaining Requests in .http Files

A request can be named with `# @name`, and a later request can pull a value straight out of its response — the REST Client's equivalent of Chapter 3/4's Postman chaining, expressed as file syntax instead of a click or a script.

```
# @name login
POST {{baseUrl}}/login
Content-Type: application/json

{ "email": "dana@example.com", "password": "..." }

###

GET {{baseUrl}}/me
Authorization: Bearer {{login.response.body.$.token}}
```

The second request never runs in isolation — it always uses whatever token the named `login` request's response actually contained, the moment it was last sent.

Thunder Client: A Postman-Like GUI, Inside VS Code

Thunder Client is a different extension entirely — a full Postman-style GUI (collections, environments, a visual request builder) running directly inside VS Code, for developers who want Postman's visual workflow without leaving the editor or maintaining a separate application.

REST Client (.http Files) vs. Thunder Client

> **REST Client**
>
> Plain text, lives in the repo, reviewable in a diff — closer to "requests as code" than "requests as a GUI collection."

> **Thunder Client**
>
> A visual, Postman-like interface inside VS Code — collections and environments managed through the GUI, not hand-written text.

Syntax	Meaning
`###`	Separates multiple requests in one file
`@name = value`	Defines a variable
`{{name}}`	References a variable
`# @name label`	Names a request so its response can be referenced later
`{{label.response.body.$.field}}`	Pulls a field out of a named request's response

💻 Coding Challenges

Challenge 1: Convert a Postman Request to .http

Convert this Postman request into `.http` file syntax: `POST {{baseUrl}}/orders` with an `Authorization: Bearer {{authToken}}` header and a JSON body `{ "productId": "P200", "quantity": 1 }`.

Goal: Practice the exact syntax — method/URL line, header line, required blank line, then body.

→ Solution

Challenge 2: Chain a Named Request

Write a two-request `.http` file: name the first request `createOrder` (returns `{ "id": "O5521" }`), then have a second request GET that order's status using the id from the first response.

Goal: Practice the `# @name` and `{{label.response.body.$.field}}` chaining syntax on a scenario other than the chapter's login example.

→ Solution

Challenge 3: REST Client or Thunder Client?

A team wants API request definitions reviewed in pull requests, diffable in Git history, with no proprietary export format involved. Which tool fits better, and why does the other one fall short for this specific requirement?

Goal: Practice applying the chapter's REST Client vs. Thunder Client comparison to a concrete team requirement.

→ Solution

⚠️ Gotcha: "Commit Your Requests" Doesn't Mean "Commit Every Value in Them"

The whole appeal of this chapter is that `.http` files are plain text meant to be committed — but the `http-client.env.json` file holding real environment values can easily contain the exact same kind of sensitive data a Postman environment export does (Chapter 3's gotcha): a real API key, a production credential. The `.http` request files themselves — which just reference `{{variableName}}` — are safe and meant to be committed; the environment file holding the actual *values* for anything sensitive should be `.gitignore`'d, exactly like a `.env` file, even though the whole point of this tool is version control.

🎯 What's Next

The next chapter leaves the editor entirely: **cURL Fundamentals** — the terminal-native tool that works anywhere a shell does, no GUI or extension required.

💻 cURL Fundamentals

Chapter 1 named cURL as the tool for when there's no GUI at all — SSH sessions, shell scripts, CI runners. This chapter builds the same requests as Chapters 2–5, this time as single terminal commands that work identically on any machine with a shell.

The Basic Shape of a cURL Command

With no options at all, `curl` sends a `GET` request and prints the response body to the terminal:

```
curl https://api.example.com/products/42
```

Specifying the HTTP Method

The `-X` flag sets the method explicitly — needed for anything other than a plain `GET`:

```
curl -X POST https://api.example.com/products
curl -X DELETE https://api.example.com/products/42
```

Headers

`-H` adds a header — one `-H` per header, repeated as many times as needed:

```
curl -H "Authorization: Bearer eyJhbGci..." -H "Accept: application/json" https://api.example.com/me
```

Sending Data: -d / --data

`-d` attaches a request body — and using `-d` at all implicitly switches the method to `POST` if `-X` wasn't specified:

```
curl -X POST https://api.example.com/products \
 -H "Content-Type: application/json" \
 -d '{"name": "Wireless Mouse", "price": 24.99}'
```

Seeing What's Actually Happening: -i and -v

By default, `curl` only prints the response **body**. Two flags reveal more: `-i` includes the response **headers** above the body (status code included); `-v` (verbose) shows the *entire* exchange — the request curl actually sent, connection details, and the full response — the closest cURL equivalent to Postman's full request/response inspection.

Default Output vs. -i vs. -v

> **Default**
>
> Just the response body — no status code, no headers visible at all.

> **-i**
>
> Status line and response headers, then the body — enough to check "did this actually succeed" without full verbosity.

`-v` goes further still, printing the request headers curl sent (prefixed with `>`) and the response headers received (prefixed with `<`) — invaluable when a request behaves unexpectedly and you need to confirm exactly what was actually sent, not just what you intended to send.

Following Redirects: -L

By default, `curl` does **not** follow a `3xx` redirect — it prints the redirect response itself (a `301`/`302` with a `Location` header) and stops there. The `-L` flag tells curl to follow it automatically and fetch the final destination instead.

```
curl -i https://example.com/old-path      # shows the 301 itself, doesn't follow it
curl -iL https://example.com/old-path     # follows the redirect, shows the final response
```

Basic Auth: -u

`-u user:password` handles HTTP Basic Authentication — curl base64-encodes the credentials and sets the `Authorization: Basic ...` header automatically, no manual encoding required.

```
curl -u admin:secretpass https://api.example.com/admin/stats
```

Flag	Purpose
`-X METHOD`	Set the HTTP method explicitly
`-H "Header: value"`	Add a request header (repeatable)
`-d / --data`	Attach a request body; implies POST if no `-X` given
`-i`	Include response status line and headers in the output
`-v`	Verbose — show the full request and response, including connection details
`-L`	Follow redirects automatically
`-u user:pass`	Send HTTP Basic Authentication credentials

Putting It Together

```
curl -iL -X POST https://api.example.com/products \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer eyJhbGci..." \
  -d '{"name": "Wireless Mouse", "price": 24.99}'
```

One command: method, two headers, a JSON body, redirect-following, and status/headers visible in the output — everything Chapters 2–3 needed several Postman UI fields for, expressed as flags on a single line.

💻 Coding Challenges

Challenge 1: Write the cURL Command

Write a cURL command that sends a `DELETE` request to `https://api.example.com/orders/55` with an `Authorization: Bearer abc123` header, and shows the response status code in the output.

Goal: Practice combining `-X`, `-H`, and `-i` in one command.

→ Solution

Challenge 2: Diagnose Missing Output

A developer runs `curl https://api.example.com/old-report` and sees an empty response body, even though they're sure the endpoint has data. Using this chapter's material, what's the most likely explanation, and what single flag would confirm it?

Goal: Practice recognizing an unfollowed redirect as a concrete real-world symptom, not just an abstract flag definition.

→ Solution

Challenge 3: -i vs. -v

A teammate wants to confirm the exact `Content-Type` header their own script actually sent (not just what they intended to send). Should they reach for `-i` or `-v`? Justify your answer.

Goal: Practice distinguishing what each flag actually reveals — response-only versus full request-and-response.

→ Solution

⚠️ Gotcha: -d Without the Right Content-Type Sends the Wrong Format

Running `curl -d '{"name": "Wireless Mouse"}' https://api.example.com/products` looks like it's sending JSON — but without an explicit `-H "Content-Type: application/json"`, curl defaults `-d`'s Content-Type to `application/x-www-form-urlencoded`, the format HTML forms use. A JSON-expecting API may then either reject the request outright or, worse, silently misinterpret the body without erroring at all. This is the exact same failure mode Chapter 2's Postman gotcha described for a raw body sent with the wrong format dropdown — always pair `-d` with an explicit `Content-Type` header when the body is JSON, never rely on curl's default.

🎯 What's Next

The next chapter is **cURL Advanced** — bearer tokens and API keys in practice, file uploads with `-F`, piping JSON output through `jq` for readability, and saving/replaying requests as scripts.

🚀 cURL Advanced

Chapter 6 covered the flags. This chapter covers the patterns that make cURL genuinely usable day to day: real auth handling, file uploads, readable JSON output, and turning a one-off command into something saved, shared, and re-run.

Bearer Tokens & API Keys, in Practice

Chapter 6's basic auth example (`-u`) is fine for that specific auth style, but most real APIs use a bearer token or an API key instead. Both go in a header — the practical question is where the token's actual *value* comes from.

```
# Hardcoding the token directly — works, but see this chapter's closing gotcha
curl -H "Authorization: Bearer abc123realtoken" https://api.example.com/me

# Reading it from an environment variable instead — the safer default
curl -H "Authorization: Bearer $TOKEN" https://api.example.com/me
```

An API key sometimes goes in a custom header (`X-API-Key: ...`) instead of `Authorization` — check the API's documentation for which it expects. Either way, avoid putting a key in the URL as a query parameter (`?api_key=...`) if a header option exists: URLs routinely end up in server access logs, browser history, and proxy logs, all places a header's contents typically don't.

File Uploads: -F

Uploading a file needs `multipart/form-data`, a different body format from `-d`'s JSON or form-urlencoded — that's what `-F` is for. The `@` prefix tells curl to read a field's value from a file on disk rather than treating it as literal text.

```
curl -X POST https://api.example.com/upload \
 -F "file=@photo.jpg" \
 -F "caption=Sunset over the harbor"
```

Mixing `-F` with `-d` in the same request doesn't work as expected — they build genuinely different body formats, so a request is either `-d`'s style or `-F`'s multipart style, not both at once.

Piping to jq for Readable JSON

A raw JSON response prints as one dense, hard-to-read line. Piping it through `jq` — already covered for general shell use in `bash_intermediate_07` — pretty-prints it, and can filter down to just the fields that matter.

```
curl -s https://api.example.com/products/42 | jq

# Just the name field:
curl -s https://api.example.com/products/42 | jq '.name'
# Every item's price, from an array response:
curl -s https://api.example.com/orders/55 | jq '.items[] | .price'
```

The `-s` (silent) flag suppresses curl's progress meter, which would otherwise clutter the output being piped into `jq`.

Saving & Replaying Requests as Scripts

A command retyped (or found again in shell history) every time isn't much better than starting from scratch. Saving it as a small shell script makes it reusable and shareable — the cURL equivalent of Chapter 5's `.http` files.

```
#!/bin/bash
# create-product.sh
curl -X POST "$BASE_URL/products" \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN" \
 --data @product.json
```

`--data @product.json` loads the body from a separate file rather than an inline string — useful once a body gets long enough to be awkward as one quoted line. Both `$BASE_URL` and `$TOKEN` come from the environment, not hardcoded — set once, reused by any script that needs them.

An Inline Command vs. a Saved Script

Inline, Retyped Each Time

Works once, then has to be found again in shell history or retyped from memory — easy to introduce a small mistake re-copying a long command.

Saved as a Script

`chmod +x create-product.sh && ./create-product.sh` — reusable, shareable with a teammate, and can be committed to the repo like any other script.

Flag / Pattern	Purpose
`-F "field=@file"`	Multipart file upload
`--data @file.json`	Load the request body from a file instead of inline text
`-o file`	Save the response body to a file instead of printing it
`| jq`	Pretty-print and filter a JSON response
`$VAR` in a script	Reads a value from the environment rather than hardcoding it

Putting It Together: A Reusable Upload Script

```
#!/bin/bash
# upload-avatar.sh
curl -s -X POST "$BASE_URL/users/me/avatar" \
 -H "Authorization: Bearer $TOKEN" \
 -F "avatar=@$1" \
 | jq '.avatarUrl'
```

Run as `./upload-avatar.sh photo.jpg` — `$1` is the shell's first script argument, so the same script uploads any file passed to it, authenticates via an environment variable, and prints just the resulting URL, readably, via `jq`.

💻 Coding Challenges

Challenge 1: Fix the Upload Command

A developer writes `curl -X POST https://api.example.com/upload -d "file=@photo.jpg"` and the file arrives corrupted/unusable server-side. What's wrong, and what's the fix?

Goal: Practice recognizing that `-d` and `-F` build genuinely different body formats, not interchangeable syntax for "attach a file."

→ Solution

Challenge 2: Filter a Response With jq

Given a response `{ "id": 42, "items": [{ "name": "Mouse", "price": 24.99 }, { "name": "Keyboard", "price": 59.99 }] }`, write the `jq` filter to print just the array of item names.

Goal: Practice writing a `jq` filter over an array field, building on `bash_intermediate_07`'s coverage.

→ Solution

Challenge 3: Make a Script Reusable

Rewrite this hardcoded command as a script using environment variables instead: `curl -H "Authorization: Bearer abc123" https://api.example.com/orders`, so the same script works for any base URL or token without editing the file.

Goal: Practice the environment-variable substitution pattern this chapter uses for reusable scripts.

→ Solution

⚠️ Gotcha: A Real Token Hardcoded Into a Saved, Committed Script

Saving a script like Chapter 6/7's examples feels like good practice — until the token inside it is a real, working credential, and the script gets committed to a shared Git repository. This is the exact same mistake as a hardcoded database connection string or a committed API key covered elsewhere on this site: once pushed, that token is compromised, regardless of whether the file is later edited or deleted. Every script in this chapter deliberately reads `$TOKEN` from the environment rather than embedding a literal value — that's not a style preference, it's the difference between a script that's safe to commit and one that isn't.

🎯 What's Next

The next chapter is **Browser-Based API Testing** — the simplest tool of all: URL query parameters, HTML forms, and reading real requests in the DevTools Network tab.

🌐 Browser-Based API Testing

Chapter 1 named the browser as the zero-setup option — nothing to open, nothing to install. This chapter covers three escalating levels of browser-only testing: the address bar, HTML forms, and the DevTools Console's `fetch()` trick — plus a deeper look at the Network tab first introduced in `wdt_lesson_04`.

URL Query Parameters

Pasting a GET URL — query string and all — directly into the address bar sends exactly that request, and most modern browsers pretty-print a JSON response automatically in a built-in, collapsible JSON viewer.

```
https://api.example.com/products?category=electronics&limit=10
```

This is the simplest possible check for any read-only endpoint — no tool, no headers, no body, just the URL.

HTML Forms: GET and POST

A plain HTML `<form>` can send a real request without any JavaScript at all. `method="GET"` builds a query string automatically from the form's named inputs; `method="POST"` sends them as a `form-urlencoded` body.

```
<form action="https://api.example.com/search" method="GET">
  <input name="category" value="electronics">
  <button type="submit">Search</button>
</form>
```

This is genuinely limited, though: a plain HTML form can't send a JSON body, can't set custom headers like `Authorization`, and is restricted to `GET`/`POST` — no `PUT`/`PATCH`/`DELETE`. It's useful for the simplest cases, and a natural stepping stone to the next section when those limits are hit.

The DevTools Network Tab, Revisited

`wdt_lesson_04` introduced the Network tab's basics. For API testing specifically, three things matter most: filtering to **Fetch/XHR** requests (hiding images/stylesheets/fonts so only actual API calls remain visible), clicking a request to see its **Headers**/**Payload**/**Response**/**Timing** sub-tabs, and right-clicking a request for **Copy → Copy as cURL** — which turns any request the page already made into a ready-to-run cURL command, using exactly Chapter 6/7's syntax.

Full Circle: Network Tab → cURL

Right-clicking a login request the page already made and choosing "Copy as cURL" produces something like:

```
curl 'https://api.example.com/login' \
 -H 'Content-Type: application/json' \
 --data-raw '{"email":"dana@example.com","password":"..."}'
```

That's a real, runnable command using Chapter 6's exact flags — captured directly from the browser with no manual reconstruction, useful for reproducing a bug exactly as the page actually sent it.

The fetch() Console Trick

Pasting a `fetch()` call directly into the DevTools Console gives full control — any method, any header, a JSON body — without leaving the browser or opening any other tool at all.

```
fetch("https://api.example.com/products", {
 method: "POST",
 headers: {
  "Content-Type": "application/json",
  "Authorization": "Bearer eyJhbGci..."
 },
 body: JSON.stringify({ name: "Wireless Mouse", price: 24.99 })
})
 .then(r => r.json())
 .then(console.log);
```

URL Bar / Forms vs. the fetch() Console Trick

> **URL Bar / HTML Forms**
>
> Zero setup, but limited to GET/POST, no custom headers, no JSON body — fine for the simplest read-only checks only.

> **fetch() in the Console**
>
> Any method, any header, a real JSON body — the browser's own equivalent of a Postman request, still without installing anything.

Technique	Can Do	Limitation
URL address bar	Simple GET requests	No headers, no body, GET only
HTML form	GET or POST with named fields	No custom headers, no JSON body, GET/POST only
DevTools Network tab	Inspect any request the page already made; export as cURL	Observes existing requests — doesn't build new ones from scratch
fetch() in Console	Any method, headers, and JSON body	Subject to the current page's CORS restrictions (see this chapter's gotcha)

💻 Coding Challenges

Challenge 1: Pick the Browser Technique

For each, name the simplest browser-only technique from this chapter that would work: (a) checking a public GET endpoint returns data, (b) sending a POST with a JSON body and a bearer token, (c) reproducing the exact request a page just made as a cURL command.

Goal: Practice matching a testing need to the least-complex browser technique that actually satisfies it.

→ Solution

Challenge 2: Write a fetch() Call

Write the `fetch()` console snippet to send a `DELETE` request to `https://api.example.com/orders/55` with an `Authorization: Bearer abc123` header, logging the parsed JSON response.

Goal: Practice the full `fetch()` options object — method, headers, and the promise chain.

→ Solution

Challenge 3: Explain the HTML Form Limitation

A developer tries to build an HTML form that sends a JSON body with a custom `X-API-Key` header. Explain why a plain `<form>` can't do this, and what technique from this chapter can.

Goal: Practice articulating specifically what HTML forms cannot do, not just that "forms are limited."

→ Solution

⚠️ Gotcha: fetch() in the Console Is Still Subject to CORS

cURL and Postman aren't browsers, so neither one enforces CORS at all — a cross-origin request that works perfectly in cURL can fail with a CORS error when the exact same `fetch()` call runs from a page's own DevTools Console, because CORS (covered in `browser_lesson_05`) is a **browser-enforced** security mechanism tied to whatever page is currently open, not a property of the request itself. A `fetch()` call to an API that doesn't send the right `Access-Control-Allow-Origin` header will be blocked in the console even though the identical request succeeds from a terminal — this is one of the most common sources of "it works in Postman/curl but not in the browser" confusion, and it's a browser restriction working as designed, not a bug in the API or in `fetch()`.

🎯 What's Next

The final chapter is the **Capstone: Testing a Real API Across All Four Tools** — a Postman collection, a VS Code `.http` file, cURL scripts, and browser testing, all against one real API, plus automating a subset of it in CI.

🏁 Capstone: Testing a Real API Across All Four Tools

Chapter 1 previewed the same request across all four tool categories. This capstone finishes what that preview started — a small, real **Task API** (login, create, get, delete), tested end to end with Postman, a VS Code `.http` file, cURL scripts, and the browser, then a subset of it wired into CI.

The API We're Testing

Endpoint	Purpose
`POST /login`	Returns a bearer token
`POST /tasks`	Creates a task, returns its id
`GET /tasks/:id`	Fetches one task
`DELETE /tasks/:id`	Deletes a task

Step 1: Postman (Chapters 2–4)

A collection with an `Auth` folder (Login) and a `Tasks` folder (Create/Get/Delete), a `baseUrl` environment variable, and the Login request's **Test script** automatically extracting the token:

```
pm.test("Login succeeded", () => pm.response.to.have.status(200));
pm.environment.set("authToken", pm.response.json().token);
```

Create Task's Test script does the same for the new task's `id`, so Get/Delete Task can reference `{{taskId}}` directly — the exact chaining pattern from Chapter 3, automated per Chapter 4.

Step 2: VS Code .http File (Chapter 5)

The same flow, committed to the repo as plain text:

```
# @name login
POST {{baseUrl}}/login
Content-Type: application/json

{ "email": "dana@example.com", "password": "..." }

###

# @name createTask
POST {{baseUrl}}/tasks
Authorization: Bearer {{login.response.body.$.token}}
Content-Type: application/json

{ "title": "Write capstone notes" }

###

GET {{baseUrl}}/tasks/{{createTask.response.body.$.id}}
Authorization: Bearer {{login.response.body.$.token}}
```

Step 3: cURL Scripts (Chapters 6–7)

A small script suite, reading secrets from the environment, never hardcoding them, chaining values via `jq`:

```
#!/bin/bash
# login.sh — logs in, exports TOKEN for the other scripts
TOKEN=$(curl -s -X POST "$BASE_URL/login" \
  -H "Content-Type: application/json" \
  -d "{\"email\":\"$EMAIL\",\"password\":\"$PASSWORD\"}" \
  | jq -r '.token')
export TOKEN

# create-task.sh — creates a task, prints its id
curl -s -X POST "$BASE_URL/tasks" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"title": "Write capstone notes"}' \
  | jq -r '.id'
```

Step 4: Browser Testing (Chapter 8)

Since creating a task needs a JSON body and a bearer token, a plain form can't do it — this needs the `fetch()` console trick:

```
fetch("https://api.example.com/tasks", {
  method: "POST",
  headers: { "Authorization": "Bearer " + token, "Content-Type": "application/json" },
  body: JSON.stringify({ title: "Write capstone notes" })
}).then(r => r.json()).then(console.log);
```

Then opening the Network tab, finding that request, and using "Copy as cURL" — comparing the exported command against Step 3's own hand-written script is a genuine way to confirm both are sending the identical request.

Step 5: Automating a Subset in CI

Not everything from Steps 1–4 belongs in CI. Browser testing (Step 4) needs a real interactive browser session and isn't meant to run unattended; Postman's collection, run headlessly through **Newman** (Chapter 4), and the cURL scripts (Step 3) both *can* run in a CI job with no human present — these are the two candidates for automation.

```
# CI step — fails the build if any assertion fails (pipelines1-4)
newman run task-api.postman_collection.json -e ci.postman_environment.json
```

Tool	Role in This Capstone
Postman	Building and organizing the flow interactively; automated in CI via Newman
.http file	The same flow, committed and reviewable in the repo
cURL scripts	Terminal-native, reusable, scriptable — usable in CI too
Browser (fetch/Network)	Manual exploration and request verification — not run in CI

💻 Coding Challenges

Challenge 1: Add the Delete Step

Extend Step 3's cURL scripts with a `delete-task.sh` that deletes the task created by `create-task.sh`, using the same `$TOKEN` environment variable.

Goal: Practice extending an existing reusable script suite consistently, rather than writing an unrelated one-off command.

→ Solution

Challenge 2: Decide What Belongs in CI

Of this capstone's four tools, which should run in an automated CI pipeline, and which shouldn't? Justify each answer using this chapter's Step 5 reasoning.

Goal: Practice applying the manual-vs-automated distinction from Chapter 1 to a concrete four-tool scenario.

→ Solution

Challenge 3: Trace a Bug Across Tools

A teammate reports "Create Task fails in our .http file but works fine in Postman." Using this course's material, list two genuinely different things worth checking before assuming it's an API bug.

Goal: Practice using multiple tools to isolate whether a discrepancy is caused by the tool's own state (e.g. stale variables) rather than the API itself.

→ Solution

⚠️ Gotcha: Four Tools Means Four Separate Copies of "Current State"

Postman's environment, the `.http` file's `http-client.env.json`, and the shell's exported `$TOKEN`/`$BASE_URL` are three **independent** stores of the same conceptual values — nothing keeps them automatically in sync. A token refreshed in Postman doesn't update the shell's `$TOKEN`; a `baseUrl` changed in the `.http` environment file doesn't touch Postman's. This is easy to forget once several tools are in play side by side, and it produces a specific, confusing symptom: the exact same request behaving differently depending on which tool ran it — not because the API changed, but because one tool's copy of "current state" quietly drifted from another's. When two tools disagree, checking whether their variables actually still match is often the fastest first step, before suspecting the API itself.

🎉 Course Complete

That's the full **API Testing & Tooling** course — from why testing matters and the tools landscape, through Postman, VS Code, cURL, and the browser, to a real API tested across all four and a subset automated in CI. Together with **API Types & Design**, this completes the first two courses in the site's APIs & Integration area; **gRPC** remains next on the bucket list.