

How to use an API

Reading time: ~15 min · Prereqs: Lessons 01–05 · 3 interactive demos inside

Why this lesson exists

Everything so far has been about using AI as a human talking to a chat window. That's great — it's where most people start and where they spend most of their time. But there's a moment, usually around "I want to do this same thing to a hundred emails," or "I want to build a small tool for my team," when the chat window stops being the right doorway.

That moment is the API. This lesson explains what it is, how to get access to it, how much it costs, and shows you the exact code that makes it go — including the version that runs on your own machine for free.

1. What an API actually is

API stands for Application Programming Interface. The name sounds technical but the idea is simple: it's a way for your code to talk to another piece of software, just like you type into a chat window, but automatically and at scale.

Here's the analogy that makes it click: imagine a restaurant. The dining room is the chat interface — you sit down, order one thing at a time, a person brings it to you. Now imagine the kitchen has a small window in the wall where you can pass order tickets directly. You can pass one ticket, or a thousand tickets, and each one comes back with a completed dish. You can process them automatically. You don't need a waiter. *That's the API — the kitchen window.*

Same restaurant. Same chef. Same food. Just a different, programmable doorway.

For AI specifically, the API lets your program send a prompt and get back a response, exactly like the chat window does, but without you sitting there typing. You can run this in a loop, at any hour, as many times as you like.

2. Getting an API key

To use a cloud AI API, you need an **API key** — a long secret string of letters and numbers that identifies your account. Every call you make is billed to it. Think of it as your credit card number and your password combined into one string.

Treat your API key like a credit card number

Seriously. If someone gets your API key, they can make calls that bill *your* account. Keys get leaked on GitHub every day because people paste them directly into code. Follow these rules from day one:

- Never paste your key into a chat window, a forum post, or an email.
- Never put it directly in your code. Not even in a comment.
- Never commit a file that contains it to version control (GitHub, GitLab, etc.).
- Store it in an **environment variable** — a named setting on your computer that your code can read without the key ever appearing in the code itself.
- Set a **spending limit** in your provider's dashboard immediately after creating the key. If something goes wrong, the bill stops at that limit.

For Claude, you create a key at **console.anthropic.com**. Sign up or sign in, click *API Keys* in the sidebar, then *Create Key*. Copy it the moment it's shown — it won't be displayed again. Then go to *Limits* and set a monthly spending cap.

Other providers work the same way. OpenAI has *platform.openai.com*. Google Cloud has AI Studio. The steps are similar everywhere: create an account, create a key, store the key safely.

One thing that might surprise you: local models like Ollama don't need a real API key at all. They're running on your own machine, so there's no account to bill. You still pass an `api_key` parameter in the code (the SDKs require the field), but any string works — even just the word `"ollama"`.

3. Your first call

The simplest way to call an API is from the terminal, using a tool called `curl` that comes pre-installed on most computers. Here's what calling Claude looks like:

```
curl https://api.anthropic.com/v1/messages \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "content-type: application/json" \
  -d '{
    "model": "claude-sonnet-4-6",
    "max_tokens": 1024,
    "messages": [
      {"role": "user", "content": "Explain what an API is in one sentence."}
    ]
  }'
```

Notice `$ANTHROPIC_API_KEY`: that's reading the key from an environment variable. The key never appears in the command itself. You'd set this variable once on your machine (`export ANTHROPIC_API_KEY=sk-ant-...` on Mac/Linux) and then reuse it everywhere.

You send that request. You get a JSON response back containing the model's reply. That's the entire thing. Everything else — Python programs, web apps, automation scripts — is just this same idea wrapped in more convenient packaging.

4. The same thing in Python

Most people don't call APIs with curl for long. They write a small program, usually in Python, because then they can put the call inside a loop, store results in a file, and do something useful at scale. Install the Anthropic library once with `pip install anthropic`, and then:

```
import anthropic

client = anthropic.Anthropic() # reads key from environment automatically

message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "Write a friendly 2-line welcome email."}
    ],
)
```

```
print(message.content[0].text)
```

That's a complete working AI program. Seven lines. Add a loop around it and a list of inputs and you can process a whole spreadsheet in the time it would take you to type the first message into the chat window.

5. Running a local model the same way

Here's one of the nicer things about Ollama and LM Studio (from Lesson 04): they both expose a local API server on your machine that follows the same format as OpenAI's API. So you can write code almost identical to the cloud version, and it runs entirely offline, at no cost, with no data leaving your computer.

With Ollama running in the background, the code looks like this:

```
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:11434/v1",  # your local Ollama server
    api_key="ollama",                    # any string works locally
)

resp = client.chat.completions.create(
    model="llama3.2",
    messages=[{"role": "user", "content": "Say hello in three languages."}],
)

print(resp.choices[0].message.content)
```

For LM Studio, the only change is the port number — its server runs on `1234` by default instead of `11434`. Everything else is identical.

This is worth pausing on. The code shape — create a client, call a method, get text back — is the same whether you're calling Claude in the cloud or a local Llama model on your laptop. The address changes, the model name changes. The shape doesn't. Learn it once and it works everywhere.

Want to see all three side by side with a copy button? Here they are:

[The three ways to call an AI API](#)

```
curl https://api.anthropic.com/v1/messages \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-H "content-type: application/json" \
-d '{
  "model": "claude-sonnet-4-6",
  "max_tokens": 1024,
  "messages": [
    {"role": "user", "content": "Explain what an API is in one sentence"}
  ]
}'
```

The shape of the call is identical across all three — the only differences are the address you send to and the SDK you use.

6. Tokens and what things cost

Cloud AI APIs don't charge per message. They charge per **token** — a small chunk of text, roughly $\frac{3}{4}$ of a word. The tokenizer in the model reads your prompt one token at a time and generates a response one token at a time. You pay for both: the tokens *you send* and the tokens *the model sends back*.

This is billed per million tokens, so each individual call costs a tiny fraction of a cent. A short conversation might use a few hundred tokens — that's less than a thousandth of a dollar. The math changes fast when you run a loop over thousands of items, which is why knowing the price and setting a spending limit matters.

The $\frac{3}{4}$ -word rule

Roughly **1 token $\approx \frac{3}{4}$ of a word ≈ 4 characters**. Common words are usually 1 token. Rare or long words might be 2–3 tokens. Punctuation, spaces, and code symbols add tokens too. The widget below will show you how this scales with real text.

Token counter

Paste any text and see a live estimate. Or use the preset buttons to see how typical inputs scale.

Short email

Article paragraph

Long contract section

Paste any text here to count tokens...

0

WORDS

0

TOKENS (APPROX)

0

CHARACTERS

Roughly 1 token $\approx$ $\frac{3}{4}$ word $\approx$ 4 characters. AI bills in tokens both ways — your prompt AND its reply.

Notice how the "long contract section" preset might look like a few sentences but clocks in at several hundred tokens. Now imagine running that through an API loop for every contract in a law firm's archive. The token count — and the bill — adds up fast.

Two practical habits that prevent surprises:

1. **Start with the cheapest model that does the job.** The gap in capability between a \$1/M-token model and a \$15/M-token model is real, but it's often not relevant for simple tasks like classification, summarisation, or extraction. Use the heavy model when you need nuanced reasoning; use the light model for bulk work.
2. **Set a spending limit before you run anything in a loop.** Every major provider lets you set a monthly cap. If your code has a bug and calls the API ten thousand times instead of ten, the limit stops the damage.

7. How much does it actually cost?

Prices vary by provider and model. The widget below lets you calculate a realistic estimate for a specific use case. The "per 1000 calls" projection is especially useful

when you're thinking about automation at scale.

API cost calculator

Model

Claude Sonnet — \$3.00/M input · \$15.00/M output

Input tokens per call

1000

Output tokens per call

500

Number of calls for batch estimate

1000

Cost per call	\$0.0105
Cost for batch	\$10.50
Input tokens (batch total)	1,000,000 tokens
Output tokens (batch total)	500,000 tokens

Prices illustrative — check provider docs for current rates. Local model cost is \$0.00 regardless of volume; hardware electricity not included.

8. Cloud vs local — picking the right tool

Now that you've seen both options, here's a plain summary of when each makes sense:

- **Cloud API (Claude, OpenAI, Gemini):** use this when you need the best quality, when setup simplicity matters more than cost, or when you're on modest hardware. Pay-as-you-go, no GPU required. Data leaves your machine.
- **Local API (Ollama, LM Studio):** use this when privacy is non-negotiable (medical notes, legal documents, anything confidential), when you have high volume that would make cloud costs impractical, or when you want to experiment without any billing risk. Needs decent hardware — 16 GB RAM is comfortable for most useful models. Data stays put.

The practical path most people take: start with the cloud (fast, no setup), switch to local if cost or privacy becomes the issue. Because both speak the same API format, the switch is usually just changing one URL and one model name.

The environment variable trick

On Mac and Linux, open your terminal and run:

```
export ANTHROPIC_API_KEY=sk-ant-your-key-here
```

Now any program you run in that terminal session can call

```
os.environ.get("ANTHROPIC_API_KEY")
```

 in Python and get it without the key ever touching your code. To make it permanent, add that line to your `~/.zshrc` or `~/.bashrc` file. On Windows, use System Properties → Environment Variables.

9. The shape is the same everywhere

The biggest insight from this lesson is structural: however you call an AI API — curl, Python, JavaScript, a no-code tool like Zapier or Make — the shape of the interaction is always the same:

1. Send a message (your prompt, plus some settings like model and max tokens)
2. Get a reply (the model's response text, plus usage stats)
3. Do something with the reply (print it, save it, feed it into the next step)

Understand that shape and you can read any AI API code, in any language, and understand what it's doing. The vocabulary might change slightly (messages vs completions, content vs choices), but the structure doesn't. That's why learning one API transfers so directly to others.

Quick-reference: the five things to do before your first real call

1. Store your key in an environment variable (never in code)
2. Set a monthly spending limit in the provider dashboard
3. Choose the cheapest model that will realistically do the job

4. Set `max_tokens` so a runaway call can't return a novel
5. Test with a single call before looping over thousands of inputs

Up next

The **homework** for this lesson asks you to explore the Anthropic Console — look around, check the Usage tab, read the pricing page — without yet creating a key. Familiarity with the interface makes the next step (actually using it) much less intimidating.

Lesson 07 takes everything from this lesson and turns it into concrete examples: what people actually build once they have an API key, what beginner-friendly first projects look like, and the pattern behind almost all of them.

LAST-REVIEWED: 2026-06-04