14

Shipping It

Streamlit UIs, FastAPI & SQLite — for a JS developer

Chapter 14 · turn a script others can touch — a UI, an API, a database. All pure Python.

14 The journey — from script to service

Your model works, but it lives in a script you run by hand. Here is the plumbing that turns it into something other people can touch — a UI to demo it, an API to serve it, a database to remember it.

Three tools, all in the standard toolbox, all runnable in minutes. And each has ONE safety rule worth more than the rest of the code.

14

Streamlit — a web UI in ~15 lines

You write Python; it draws widgets in a browser

Streamlit turns a plain Python script into a web app — no HTML, no JavaScript, no React, no separate frontend. The fastest way to put a model in front of a stakeholder is a text box they can type into.

```
import streamlit as st

st.title('Review Sentiment Demo')
review = st.text_area('Paste a review:')
if st.button('Analyze'):
    label = 'positive'
    st.write(f'Sentiment: {label}')
```

Line by line

st.title → a heading
st.text_area → an input box; returns the typed text as a plain string
st.button → True only on the run it was just clicked
st.write → shows anything: text, number, table, chart

```
Run it:   python -m streamlit run app.py      # starts a local server + opens a browser tab
```

14

The re-run model — the thing to grasp

Every widget touch re-runs your whole script

Streamlit's one big idea: every time the user touches ANY widget — a click, a keystroke, a slider drag — Streamlit re-runs your entire script top to bottom, fresh. Sharp consequence: ordinary variables do NOT survive between runs. A count = 0 you increment on a click resets to 0 on the very next run.

```
if 'messages' not in st.session_state:
    st.session_state.messages = []    # runs ONCE per
session

prompt = st.chat_input('Say something')
if prompt:
    st.session_state.messages.append(prompt)

for msg in st.session_state.messages:
    st.write(msg)
```

To keep values across re-runs, use st.session_state — a dictionary that lives for the whole browser session.

The if 'messages' not in st.session_state guard is the pattern to recognise: it sets up the history exactly once. Every later run reuses the stored list, so the chat accumulates instead of resetting.

14

Reading & judging Streamlit

Where it belongs — and the caching trap

Red flag 1 — wrong home

Streamlit is for prototypes, internal tools and demos — NOT customer-facing production. A Streamlit app on a public customer path is a red flag; that job belongs to React / Next with a real backend.

Red flag 2 — no caching

Since the script re-runs on every keystroke, expensive work (a model load, an API call, a big query) written to run each time means it reloads on EVERY keystroke. Good code caches it.

```
@st.cache_resource                # load the model ONCE, reuse across every re-run
def load_model(): ...             # (@st.cache_data for query results; or stash in
session_state)
```

14

FastAPI from zero

A UI is for humans; an API is for programs

Other services, front-ends and agents call your model over HTTP. FastAPI is the standard Python framework for this — fast, modern, and it validates requests for you.

```
from fastapi import FastAPI

app = FastAPI()

@app.get('/')
def home():
    return {'status': 'ok', 'service': 'sentiment'}
```

Reading it

app = FastAPI() → creates the app object
@app.get('/') → a decorator: 'on an HTTP GET to the path /, call this function'

The function returns a plain dict — FastAPI turns it into JSON for you. No serialization code to write.

```
Run it:   fastapi dev main.py      # or:  uvicorn main:app --reload   (main = file, app = object)
```

14

Query parameters, typed for free

The type hints ARE the validation contract

The simplest input is a query parameter — the `?key=value` part of a URL. FastAPI reads them straight from your function's arguments.

```
@app.get('/margin')
def margin(revenue: float, expense: float):
    profit = revenue - expense
    return {'profit': profit, 'margin_pct': profit / revenue * 100}

# a request to  /margin?revenue=100&expense=40
#   =>  {'profit': 60.0, 'margin_pct': 60.0}
```

Because you annotated `revenue: float`, FastAPI pulls that value out of the URL, converts the text '100' into the number 100.0, and validates it. Send `/margin?revenue=abc` and FastAPI rejects it with a clear 422 error BEFORE your function ever runs. The type hints are not decoration — they are the contract.

14

GET vs POST, and a request body

Small input in the URL; big or private in the body

GET — fetch things

Input rides in the URL. Should not change server state. The URL shows up in logs and history — so no large or sensitive payloads. A one-line review fits.

POST — send data to process/store

Data travels in a request BODY, separate from the URL. A 500-word document or a chat history belongs here, not in a query string.

```
class Review(BaseModel):        # a Pydantic model: a
class with typed fields
    text: str

@app.post('/analyze')
def analyze(review: Review):
    return {'length': len(review.text)}
```

The client sends JSON like {'text': 'I want a refund'}. Because the parameter is typed as Review, FastAPI reads the body, parses the JSON, checks that text is present and is a string, and hands you a ready Review object.

Missing or malformed? The caller gets a precise error — you never write a single if 'text' not in data check.

14

The docs page comes free

An interactive API explorer, generated for you

Visit `/docs` in the browser

FastAPI has generated a full Swagger UI — an interactive page listing every endpoint, its parameters and its expected body. Each one has a 'Try it out' button that fires a real request from the browser.

It is built from your type hints and Pydantic models automatically. Nothing to maintain, always accurate. (Postman is the other common tool for testing by hand.)

JS → Python

FastAPI is roughly Express, with two things baked in that you would bolt on in Node:

1. Validation is declarative through Pydantic type hints — as if Zod were part of the framework and wired to every route.

2. The OpenAPI / Swagger docs are generated, so there is no hand-written spec drifting out of sync.

Reading & judging: good endpoints TAKE and RETURN Pydantic models — validated boundaries where bad input is rejected up front. An endpoint that accepts a raw dict and reaches into it with `data['text']` has thrown away the framework's best feature and will fail ugly on bad input.

14

SQLite — a real database, zero setup

A whole relational database in one file

To remember results across restarts you need a database. Python ships one: sqlite3 — real SQL, real tables, stored in a single file on disk. No server to install, no connection string, no Docker.

Newcomer trap: do NOT run pip install sqlite3. It is part of the standard library — there is nothing to install. You just import sqlite3.

```
conn = sqlite3.connect('shop.db')    # creates the file
if missing
cur = conn.cursor()
cur.execute(
  'CREATE TABLE IF NOT EXISTS reviews ('
  '  id INTEGER PRIMARY KEY, text TEXT, label TEXT, score
REAL)')
conn.commit()
```

conn = the connection to the file.
cur = a cursor — the object you run SQL through.

CREATE TABLE IF NOT EXISTS runs once and is safe to run twice — it won't error the second time.

score REAL = a floating-point number.

14

Inserting — with placeholders, always

The single most important safety rule here

To put values into a query, use the ? placeholder and pass the values as a separate tuple. Never glue values into the SQL text.

```
cur.execute(
  'INSERT INTO reviews (text, label, score) VALUES (?,
?, ?)',
  (text, label, score),
)
conn.commit()           # <-- without this, nothing is
saved
```

Each ? is a slot. The tuple fills the slots in order — and SQLite treats those values strictly as DATA, never as SQL code.

A review containing '; DROP TABLE reviews;-- is stored as harmless text, not executed.

Note conn.commit(). SQLite runs your writes inside a transaction — they are not durably saved until you commit. Skip it, close the connection, and your inserts silently vanish. This trips up nearly everyone once. (cur.executemany takes a list of tuples to insert many rows at once.)

14

Reading it back — and closing with 'with'

Same cursor, same ? rule

```
cur.execute(
  'SELECT id, text, score FROM reviews WHERE label
= ?',
  ('negative',),          # filter value goes in as
a tuple, too
)
negatives = cur.fetchall()
for row in negatives:
    print(row)       # each row is a tuple of the
columns
```

```
with sqlite3.connect('shop.db') as conn:
    cur = conn.cursor()
    cur.execute(
      'INSERT INTO reviews VALUES (?, ?, ?,
?)',
      (None, 'Late', 'negative', 0.7))
    # commit happens on a clean exit
conn.close()
```

fetchall() returns a list of rows, each a tuple of column values in the order you selected them (fetchone() gets a single row). The connection is also a context manager: the `with` block commits automatically on a clean exit and rolls back if an exception is raised inside — so a stray return can't lose your write. It manages the transaction, not the connection, so you still call conn.close() after.

14

Reading & judging SQLite

One cardinal sin, one quiet bug, one graduation

Cardinal sin — string-built SQL

Building SQL by string formatting — f'... WHERE label = {user_input}' or '...' + value — is a SQL-injection hole.

With LLM output flowing into your database it is doubly dangerous: model text is untrusted input, just like user text. Always use ? placeholders and pass a tuple.

The quiet bug — a missing conn.commit(). Code that 'works' in the same session but loses everything on restart.

Know when to graduate. SQLite is superb for prototypes, single-writer tools and local persistence. For many concurrent writers, real schema migrations, or a shared database across services — move to Postgres with an ORM like SQLAlchemy.

You have just stored LLM sentiment results in a real database and pulled back the ones the model was most confident about.

14

Calling other APIs — the HTTP clients

Serving an API is only half the story

Often your code also CALLS other HTTP APIs — an LLM provider, a data service. In Node you reach for fetch. Python gives you two friendly equivalents.

```
requests  —  the quick script

requests.get(url).json()   → parsed JSON
resp.status_code   → the HTTP status
params={'q': 'hello'}   → query params

Synchronous and beautifully simple.
```

httpx — the guide's default

Same friendly API as requests, PLUS async / await support — so it fits inside FastAPI's async endpoints and won't block your server on a slow upstream call.

Rule of thumb: requests for a quick script · httpx for anything that runs inside a server or needs concurrency.

14 The three tools, and the bugs to hunt

The recap

Streamlit → a demo UI, written entirely in Python

FastAPI → an HTTP API with validation and docs for free

SQLite → a zero-setup file database that remembers results

All pure Python. All runnable in minutes.

Bugs to hunt when reading:

1. SQL built by string-formatting — an injection hole. Use ? and a tuple.

2. A missing conn.commit() — works now, loses data on restart.

3. Expensive work re-running on every Streamlit keystroke — cache it.

4. An endpoint taking a raw dict instead of a Pydantic model.

The teaching — shipping a model is plumbing, not magic: a Streamlit box to demo it, a FastAPI endpoint to serve it, a SQLite file to remember it. Each has one safety rule worth more than the rest of the code.