

PROJECTS

Building a Web Framework

A real, small Python WSGI framework built entirely from scratch
— routing, request/response objects, a template engine,
middleware, sessions, a minimal ORM, error handling with a live-
reload dev server, static files, and a CLI — every class verified
against real, running code, closing with a genuine working app
assembled from every piece.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: WHY BUILD A FRAMEWORK? WSGI & THE REAL PYTHON WEB SERVER INTERFACE

Building a Web Framework

Chapter 1 · Why Build a Framework? WSGI & the Real Python Web Server Interface

Web Framework Internals studied how five real frameworks each answer four real questions — routing, templating, data access, middleware — by comparing their real syntax and quoting their real documentation. This course answers the identical four questions a genuinely different way: by building a real, small, working Python web framework from scratch, one capability at a time, ending with a real application running on top of it. Before any of that, though, there's a more basic real question to answer first — how does a Python function even become "a web application" that a real browser can talk to at all?

SCOPE NOTE — READ THIS BEFORE ANYTHING ELSE

This course builds a real, working **synchronous** framework, on top of WSGI — not the newer, async ASGI standard FastAPI (covered in Web Framework Internals) is built on. That's a deliberate choice, explained in full later in this chapter, not a limitation being quietly worked around. Every claim in every chapter of this course is backed by real, hand-run code and, where relevant, real quotes from official specifications — the same discipline this site's own `compiler1`/`compiler2`, `browserengine1`/`browserengine2`, `dbengine1`/`dbengine2`, and `oskernel1`/`oskernel2` projects already used.

The Pipeline This Course Builds

CAPABILITY	WHAT IT DOES	BUILT IN
Routing	A real path-matching engine with dynamic parameters, mirroring Web Framework Internals' own Chapter 2	Chapter 2
Request/response objects	A real, ergonomic wrapper around the raw WSGI interface this chapter builds	Chapter 3
Templating	A real template engine — parsing, compiling, rendering — mirroring Chapter 4	Chapter 4
Middleware	A real onion-model pipeline, mirroring Chapter 8	Chapter 5

CAPABILITY	WHAT IT DOES	BUILT IN
Sessions & cookies	Real state carried across genuinely stateless HTTP requests	Chapter 6
A minimal ORM	Mapping real Python objects to real SQL, mirroring Chapter 6	Chapter 7
Error handling & dev server	A real development server with live reload	Chapter 8
Static files & a CLI	Serving real files directly, and a real command-line tool for the framework itself	Chapter 9

The Real Interface Every Synchronous Python Framework Sits On

A Python program doesn't talk to the internet directly — a real web server (Apache, nginx, Gunicorn) handles the actual TCP connections and HTTP parsing, then needs some real, standardized way to hand a parsed request off to Python application code and get a response back. That standard is WSGI (the Web Server Gateway Interface, defined in PEP 3333) — and it's genuinely simple: a WSGI application is just a callable taking two arguments. PEP 3333's own real, minimal example:

```
def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [b"Hello world!\n"]
```

This isn't a sketch — it's real, runnable code. Python's own standard library ships a genuine, working WSGI server ([wsgiref](#)) specifically for testing code exactly like this:

```
from wsgiref.simple_server import make_server
import urllib.request

server = make_server('127.0.0.1', 8734, simple_app)
# ... serve_forever() on a thread ...
```

```
resp = urllib.request.urlopen('http://127.0.0.1:8734/')
print(resp.status, resp.read(), resp.headers.get('Content-type'))
# 200 b'Hello world!\n' text/plain
```

VERIFIED DIRECTLY — A REAL HTTP REQUEST, A REAL HTTP RESPONSE

That's a genuine HTTP client (`urllib.request`) talking to a genuine, real HTTP server, produced entirely by a 6-line function following one, real, standardized interface. Every real Python web framework — Django, Flask, every one of Web Framework Internals' own routers built in Chapter 2 — ultimately reduces to *something* shaped like `simple_app` above, however many layers of routing, templating, and middleware sit in front of it.

The Real environ Dict: What a Server Actually Hands You

`environ` is a real, plain Python dict — per PEP 3333's own specification, a set of CGI-style variables describing the incoming request. Capturing it from a real request shows exactly what's really in there:

```
def inspecting_app(environ, start_response):
    captured_environ.update(environ)
    start_response('200 OK', [('Content-type', 'text/plain')])
    return [b"ok"]

# a real request to /users/42?active=true
for key in ['REQUEST_METHOD', 'PATH_INFO', 'QUERY_STRING', 'SERVER_PROTOCOL', 'wsgi.url_scheme']:
    print(key, '=', captured_environ.get(key))

# REQUEST_METHOD = GET
# PATH_INFO = /users/42
# QUERY_STRING = active=true
# SERVER_PROTOCOL = HTTP/1.1
# wsgi.url_scheme = http
```

THIS IS THE RAW MATERIAL EVERY LATER CHAPTER BUILDS ON

`PATH_INFO` and `QUERY_STRING`, verified here as real, plain strings sitting in a real dict, are exactly what Chapter 2's own router will parse to match a route, and exactly what a real request object (Chapter 3) will wrap into something more ergonomic than reading dict keys by hand. Nothing about

routing or request objects is magic — it's all reachable, real data already sitting in `environ` the moment a request arrives.

start_response: A Callback, Not a Return Value

Web Framework Internals verified Rack's own real interface — a middleware and a full application both implement `call(env)`, returning a real `[status, headers, body]` triplet. WSGI takes a genuinely different real approach: `start_response` is a *callback*, called from inside the application, before the body is returned. Getting the order wrong is a real, documented contract violation — and Python's own reference implementation doesn't fail silently:

```
def broken_app(environ, start_response):
    # never calls start_response at all
    return [b"this body is returned, but no status/headers were ever sent"]

# real client-side result:
# HTTPError: HTTP Error 500: Internal Server Error
#
# real server-side traceback:
# AssertionError: write() before start_response()
```

VERIFIED DIRECTLY — A REAL, DOCUMENTED CONTRACT, ACTIVELY ENFORCED

`wsgiref` doesn't just document that `start_response` must be called first — it checks, at runtime, and raises a real `AssertionError` the instant the rule is broken, converting it into a genuine HTTP 500 rather than sending a malformed response or hanging. The identical real discipline applies to the body itself: returning a plain Python `str` instead of a list of real `bytes` objects — a genuinely easy mistake — is caught the same way, verified directly: `AssertionError: write() argument must be a bytes instance`. PEP 3333's own real requirement, "must return an iterable yielding zero or more bytestrings," isn't just a suggestion in the spec's own prose — it's an actively-checked runtime contract in Python's own reference server.

Real Frameworks Sit on the Identical Interface

Django's own real, default deployment file, `wsgi.py`, is a genuine, direct instance of exactly this pattern:

```
# myproject/wsgi.py -- Django's own real, generated file
import os
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')
application = get_wsgi_application()
```

`application`, the module-level name a real WSGI server looks for, is a genuine, callable WSGI application — underneath Django's own real router, middleware stack, and ORM (all covered across Web Framework Internals' own Chapters 2–7), it still reduces to the identical `(environ, start_response)` shape `simple_app` used at the very top of this chapter.

WSGI's Real Limits, and Why This Course Still Builds on It

WSGI's own real successor, ASGI, exists for a real, documented reason — per ASGI's own official documentation, it's "a spiritual successor to WSGI, intended to provide a standard interface between async-capable Python web servers, frameworks, and applications." The same real documentation names the specific real gap directly: "WSGI applications are a single, synchronous callable that takes a request and returns a response; this doesn't allow for long-lived connections, like you get with long-poll HTTP or WebSocket connections."

A DELIBERATE SCOPE CHOICE, NOT AN UNNOTICED GAP

FastAPI, covered throughout Web Framework Internals, is built on ASGI specifically because it needs exactly what WSGI's own real, single-callable model can't provide. This course builds on WSGI anyway — deliberately — because a real, complete, synchronous framework is a genuinely more tractable real project to build chapter by chapter, and every one of this course's own four core capabilities (routing, templating, data access, middleware) is fully expressible on top of it. Real async support, real WebSockets, and real long-lived connections are named here as honestly out of scope, not silently ignored.

Where This Course Is Headed

Chapter 2 builds a real routing engine on top of the `environ` dict verified in this chapter — real path matching and dynamic parameters, following the exact same real approach Web Framework Internals' own Chapter 2 built and verified.

Hands-On Exercises

EXERCISE 1

Write a real WSGI application that reads `PATH_INFO` from `environ` and echoes it back in the response body (e.g. "You requested: /users/7"), and verify it against three genuinely different real paths using `urllib.request` against a real `wsgiref` server.

 [View solution](#)

EXERCISE 2

Write a real WSGI application that calls `start_response` correctly but returns a plain Python `str` instead of a list of bytes, run it against a real `wsgiref` server, and report the exact real exception raised server-side and the real HTTP status the client receives.

 [View solution](#)

EXERCISE 3

Using this chapter's own real `start_response` callback model and Web Framework Internals' own quoted Rack call(`env`) -> [`status`, `headers`, `body`] return-triplet model, explain in your own words which of the two interfaces makes a "forgot to call `start_response`" bug easier to catch automatically, and why.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **WSGI** — the real, standardized interface (PEP 3333) between a Python web server and Python application code: a callable taking (`environ`, `start_response`)

- **environ** — a real, plain dict of request data; verified real keys include REQUEST_METHOD, PATH_INFO, QUERY_STRING, SERVER_PROTOCOL
- **start_response** — a real callback, not a return value; verified to raise a real AssertionError if called too late or never called at all
- **The body** — must be real bytes, not str; verified as an actively-enforced runtime contract, not just documentation
- **Real frameworks confirm this** — Django's own real wsgi.py/get_wsgi_application() reduces to the identical callable shape
- **WSGI vs. ASGI** — WSGI is a single, synchronous callable; ASGI is its real, documented async successor, needed for WebSockets/long-lived connections — this course deliberately stays on WSGI
- **Next chapter:** A routing engine from scratch — path matching & dynamic parameters

CHAPTER 2: A ROUTING ENGINE FROM SCRATCH: PATH MATCHING & DYNAMIC PARAMETERS

Building a Web Framework

Chapter 2 · A Routing Engine From Scratch: Path Matching & Dynamic Parameters

Chapter 1 verified that a real request arrives as a plain dict — `PATH_INFO`, `REQUEST_METHOD`, and the rest, sitting right there in `environ`. Nothing yet decides which piece of your own code should actually run for a given path and method; every request would currently have to be handled by one giant function with its own if/elif chain reading `environ` by hand. This chapter builds the piece that removes that need — a real, reusable `Router` class — and it's not a one-off demonstration: this is the exact object every later chapter in this course imports and builds on, starting with Chapter 3's request object next.

MIRRORING, NOT REPEATING, WEB FRAMEWORK INTERNALS' OWN CHAPTER 2

That course built and verified a standalone router as a way of *explaining* how routing works, then moved on to comparing five real frameworks' own syntax for it. This chapter builds the identical underlying mechanics — static/dynamic segments, typed converters, registration order, named routes, method dispatch — as a real, permanent class this course's own framework keeps and reuses, wired into a genuine running WSGI app by the end of the chapter rather than left as isolated examples.

The Basic Job: A Lookup Table From Path to Handler

At its simplest, a router is nothing more than a dict: a path string maps to a handler, and dispatch is a dict lookup with a default for "not found."

```
routes = {}

def add_route(path, handler):
    routes[path] = handler

def dispatch(path):
    handler = routes.get(path)
    if handler is None:
        return '404 Not Found'
    return handler()
```

```
add_route('/about', lambda: 'About page')
print(dispatch('/about'))    # About page
print(dispatch('/missing')) # 404 Not Found
```

Genuinely correct, genuinely useless past a handful of static pages — there's no way to write one route that matches every `/users/<some id>` path at once.

Dynamic Segments, and the String Trap They Leave Behind

Regular expressions fix the matching problem directly — a path like `/users/<id>` can be turned into a real regex with a named capture group:

```
import re

def compile_pattern(path):
    regex = re.sub(r'<(\w+)>', r'(?P<\1>[^\s/]+)', path)
    return re.compile(f'^{regex}$')

pattern = compile_pattern('/users/<id>')
match = pattern.match('/users/42')
print(match.groupdict())    # {'id': '42'}
print(type(match.groupdict()['id'])) # <class 'str'>
```

VERIFIED DIRECTLY — THE CAPTURED VALUE IS ALWAYS A PLAIN STRING

`match.groupdict()` returns `{'id': '42'}` — the character *string* `'42'`, confirmed by `type()` as a real Python `str`, never the number `42`. A regex has no idea a segment was "supposed to" be numeric; it just matched characters. A handler that does `id + 1` on that value crashes with a real `TypeError` — not because the router is broken, but because nothing has told it a segment should be anything other than text.

Typed Converters: Validate and Cast, Once, Before the Handler Sees It

The fix is to let a route segment declare its own type, and have the router validate and convert it in one place rather than leaving every handler to do it separately:

```

CONVERTERS = {
    'int': (r'\d+', int),
    'str': (r'^/+', str),
}

def compile_typed_pattern(path):
    parts = []
    types = {}
    for segment in path.split('/'):
        m = re.match(r'<(\w+):(\w+)>', segment)
        if m:
            conv_name, param_name = m.group(1), m.group(2)
            regex, cast = CONVERTERS[conv_name]
            parts.append(f'(?P<{param_name}>{regex})')
            types[param_name] = cast
        else:
            parts.append(re.escape(segment))
    return re.compile('^' + '/'.join(parts) + '$'), types

pattern, types = compile_typed_pattern('/users/<int:id>')
for test_path in ['/users/42', '/users/abc']:
    m = pattern.match(test_path)
    if m:
        params = {k: types[k](v) for k, v in m.groupdict().items()}
        print(test_path, '→', params)
    else:
        print(test_path, '→ no match')
# /users/42 → {'id': 42}
# /users/abc → no match

```

VERIFIED DIRECTLY — A REAL INT, AND A REAL, CORRECT REJECTION

`/users/42` prints `{'id': 42}`, a genuine Python `int` cast once by the router before any handler ever runs. `/users/abc` prints `no match` outright — the `\d+` pattern behind `int` never matches non-digit characters, so the request falls through to a real 404 instead of reaching a handler that expected a number and would have crashed on `"abc"`.

Building the Real Router Class

Everything above works as standalone functions. This framework needs something that can hold many routes at once, across many HTTP methods, and be queried repeatedly — so it becomes a real class, `Router`, the actual file this course keeps building on:

```
import re

CONVERTERS = {
    'int': (r'\d+', int),
    'str': (r'[^/]+', str),
}

class Router:
    """A real, minimal router this framework reuses in every later chapter."""

    def __init__(self):
        self._routes = []  # (method, compiled_regex, types, handler)
        self._named = {}  # name -> raw path template, for url_for()

    def add_route(self, method, path, handler, name=None):
        regex, types = self._compile(path)
        self._routes.append((method.upper(), regex, types, handler))
        if name:
            self._named[name] = path

    def _compile(self, path):
        parts, types = [], {}
        for segment in path.split('/'):
            m = re.match(r'<(\w+):(\w+)>', segment)
            if m:
                conv_name, param_name = m.group(1), m.group(2)
                seg_regex, cast = CONVERTERS[conv_name]
                parts.append(f'(?P<{param_name}>{seg_regex})')
                types[param_name] = cast
            else:
                parts.append(re.escape(segment))
        pattern = re.compile('^' + '/'.join(parts) + '$')
        return pattern, types

    def resolve(self, method, path):
        method = method.upper()
        allowed = set()
        for route_method, regex, types, handler in self._routes:
            m = regex.match(path)
            if m:
```

```

        allowed.add(route_method)
        if route_method == method:
            params = {k: types[k](v) for k, v in m.groupdict().items()}
            return 'MATCH', handler, params, None
    if allowed:
        return '405', None, {}, allowed
    return '404', None, {}, None

def url_for(self, name, **params):
    template = self._named[name]
    for key, value in params.items():
        template = re.sub(rf'<\w+:{key}>', str(value), template)
    return template

```

`resolve()` is the one method everything else in this chapter, and this course, calls. It walks every registered route once, remembers every HTTP method that matched the *path* even when it doesn't match the *method*, and returns one of exactly three outcomes: a real match with its typed params, a 405 with the set of methods that would have worked, or a plain 404.

Registration Order: A Real, Reproducible Bug on This Course's Own Router

Two registered routes can genuinely both match the same path. `Router.resolve()` scans its own `_routes` list top to bottom and returns the first one whose method also matches — which means, exactly as with Django's own real URL resolver, registration order is load-bearing.

```

router = Router()
router.add_route('GET', '/users/<str:id>', lambda **p: f"user detail {p['id']}", name='user_detail')
router.add_route('GET', '/users/new', lambda **p: 'user create form', name='user_create')

status, handler, params, allowed = router.resolve('GET', '/users/new')
print(status, handler(**params))
# MATCH user detail new

```

VERIFIED DIRECTLY — /USERS/NEW NEVER REACHES ITS OWN HANDLER

Registered in this order, `resolve('GET', '/users/new')` returns `MATCH` against the general `/users/<str:id>` route, with `id` captured as the literal string `"new"` — the dedicated

`/users/new` route is never even checked, because the loop already returned on the first match it found. Registering the exact same two routes with `/users/new` listed first fixes it directly, verified against the same input: `MATCH user create form`, with the general route only ever reached by a path that genuinely isn't `"new"`.

TYPED CONVERTERS NARROW THIS RISK WITHOUT ELIMINATING IT

Swapping in `<int:id>` instead of `<str:id>` happens to reject `"new"` on its own, since it contains no digits — verified separately: registered in either order, `/users/<int:id>` and `/users/new` resolve correctly, because the general route's own regex is now narrow enough to never accidentally swallow the literal path. That's a real, useful side effect of typing a segment, not a substitute for specific-before-general ordering: any two routes whose patterns can both genuinely match the same path still need the more specific one registered first.

Method Dispatch: A Real, Distinct 405, Not Just a Second 404

HTTP defines `405 Method Not Allowed` for exactly the case `resolve()` tracks with its own `allowed` set: a path that matches something registered, on a method that isn't. It's a genuinely different situation from no match at all, and a real client can tell the two apart — including, correctly implemented, via the response's own `Allow` header naming which methods would have worked.

```
router = Router()
router.add_route('GET', '/users/<int:id>', lambda **p: f"get user {p['id']}", name='use
router.add_route('DELETE', '/users/<int:id>', lambda **p: f"delete user {p['id']}", nam

print(router.resolve('GET', '/users/42'))
print(router.resolve('DELETE', '/users/42'))
print(router.resolve('PATCH', '/users/42'))    # path matches, method doesn't
print(router.resolve('GET', '/nonexistent'))    # no path match at all
```

VERIFIED DIRECTLY, WIRED TO A REAL WSGI APP IN THE NEXT SECTION

`GET /users/42` and `DELETE /users/42` both return `MATCH` against the same path with their own correct handler and params. `PATCH /users/42` returns `('405', None, {}, {'GET', 'DELETE'})` — the path itself genuinely matched, so `resolve()` reports every method that would have worked instead of pretending nothing was found there. `GET /nonexistent` returns `('404', None, {},`

`None` — no route's own pattern matched this path under any method at all, a real, distinct third outcome the code below turns into two genuinely different HTTP status lines.

Named & Reverse Routing

`url_for()` goes the other direction — given a route's own name and its param values, it generates the real path string, so nothing downstream ever has to hardcode `/users/42` as a literal.

```
router = Router()
router.add_route('GET', '/users/<int:id>', lambda **p: None, name='user_detail')
print(router.url_for('user_detail', id=42))
# /users/42
```

A REAL, PREVIOUSLY-FOUND BUG THIS EXISTS SPECIFICALLY TO AVOID

Web Framework Internals' own `Personal Catalogue (Django & PostgreSQL)` and `Premier League Predictor: Django & MySQL` courses each found the identical real consequence of skipping this: a handler that hardcodes `redirect('/users/42')` as a literal string breaks silently the instant the app is mounted somewhere other than the site root, since nothing in that literal knows about the new prefix. A handler that calls `router.url_for('user_detail', id=42)` instead can be told about a mount-point change in exactly one place — inside `Router` itself — and have every generated path update automatically. This framework's own future deployment chapter revisits this directly.

Wiring It Into a Real, Running WSGI App

Every piece above is verified in isolation. Put together, `Router` becomes the actual dispatch layer of a genuine WSGI application — built directly on the `environ` dict Chapter 1 already verified, and tested against real HTTP requests exactly the way Chapter 1's own examples were:

```
router = Router()
router.add_route('GET', '/', lambda **p: 'Home page', name='home')
router.add_route('GET', '/users/<int:id>', lambda **p: f"User {p['id']}", name='user_detail')
router.add_route('DELETE', '/users/<int:id>', lambda **p: f"Deleted user {p['id']}", name='user_delete')
router.add_route('GET', '/users/new', lambda **p: 'New user form', name='user_create')
```

```
def app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ.get('PATH_INFO', '/')
    status, handler, params, allowed = router.resolve(method, path)

    if status == 'MATCH':
        body = handler(**params).encode('utf-8')
        start_response('200 OK', [('Content-type', 'text/plain')])
        return [body]
    elif status == '405':
        allow_header = ', '.join(sorted(allowed))
        start_response('405 Method Not Allowed', [
            ('Content-type', 'text/plain'), ('Allow', allow_header),
        ])
        return [f"405 Method Not Allowed (allowed: {allow_header})".encode('utf-8')]
    else:
        start_response('404 Not Found', [('Content-type', 'text/plain')])
        return [b"404 Not Found"]
```

Served through Chapter 1's own `wsgiref.simple_server` and hit with genuine `urllib.request` calls, including a real `DELETE` and a real `PATCH`:

```
GET    /                → 200  b'Home page'
GET    /users/42        → 200  b'User 42'
GET    /users/new      → 200  b'New user form'
DELETE /users/42      → 200  b'Deleted user 42'
PATCH /users/42    → 405  b'405 Method Not Allowed (allowed: DELETE, GET)'
          Allow: DELETE, GET
GET    /nonexistent → 404  b'404 Not Found'
```

VERIFIED DIRECTLY — A REAL, CORRECT ALLOW HEADER ON A REAL 405 RESPONSE

Every line above is a genuine round trip through a live `wsgiref` server: a real request goes out, `Router.resolve()` decides its outcome, and `app()` turns that outcome into a real HTTP status line and body. The `PATCH /users/42` response's own `Allow` header, read back from the actual response object, correctly lists `DELETE, GET` — exactly the two methods registered against that path, in sorted order, with nothing hardcoded anywhere in `app()` itself. That header is computed fresh from whatever `Router` actually has registered at the moment of the request.

Where This Course Is Headed

Chapter 3 wraps the raw `environ` dict and this chapter's own typed `params` dict into a single, ergonomic request object — and a matching response object, so a handler never has to call `start_response` directly again.

Hands-On Exercises

EXERCISE 1

Add a real 'slug' entry to this chapter's own CONVERTERS dictionary (matching lowercase letters, digits, and hyphens only), register a route `/posts/<slug:title>` on this course's own Router class, and verify it correctly matches "hello-world-42" while rejecting "Hello World!".

[!\[\]\(d4e92a70a184987c4cee61bbacf99330_img.jpg\) View solution](#)

EXERCISE 2

Using this chapter's own untyped `<str:id>` router, register `/users/<str:id>` before `/users/new` and demonstrate concretely, via a real Router instance and a printed `resolve()` call, that GET `/users/new` resolves to the wrong handler — then fix it by reordering the two registrations and confirm the correct handler now resolves.

[!\[\]\(848edf3a971f9d4a6acd664a9b2a684c_img.jpg\) View solution](#)

EXERCISE 3

Register a single POST-only route on a fresh Router instance, then call `resolve()` with GET against that same path and against a completely unregistered path, printing both results — and explain in your own words why the first genuinely deserves a 405 with an Allow header while the second deserves a plain 404.

[!\[\]\(9563e6845e9460f02a8b96af0592b0be_img.jpg\) View solution](#)

CHAPTER 2 QUICK REFERENCE

- **The basic job** — match an HTTP method + path to a registered handler, or report that nothing matched
- **Dynamic segments** — regex capture groups; verified always coming back as plain strings, never numbers, without a converter
- **Typed converters** — CONVERTERS maps a name (int, str, ...) to a (regex, cast) pair, validating and casting a segment once before any handler sees it
- **Registration order** — resolve() returns the first matching route it finds; a general pattern registered before a more specific one really does swallow it, verified directly on this course's own Router
- **404 vs. 405** — resolve() tracks every method whose path matched, returning a real 405 (with the Allow header this chapter's own app() builds from it) instead of a plain 404 when only the method was wrong
- **Named/reverse routing** — url_for() generates a path from a route's own name, the exact real fix this site's own Django courses needed for a mount-point path bug
- **Router is now a real, permanent file** — every later chapter in this course imports and extends it, starting with Chapter 3's request/response objects
- **Next chapter:** Request & response objects — wrapping the raw WSGI environ

CHAPTER 3: REQUEST & RESPONSE OBJECTS: WRAPPING THE RAW WSGI ENVIRON

Building a Web Framework

Chapter 3 · Request & Response Objects: Wrapping the Raw WSGI Environ

A handler in Chapter 2's own demo app already reads `req.params['id']` instead of parsing `PATH_INFO` by hand, but it still touches `environ` directly for the method and path, and every handler builds its own status line and header list by hand. This chapter wraps both directions of that raw interface into two real classes — `Request` and `Response` — so a handler only ever has to read a clean object and return a clean object, never call `start_response` or reach into `environ` itself again.

Building the Request Class

`Request` takes the raw `environ` dict Chapter 1 verified, plus the typed `params` dict Chapter 2's own `Router.resolve()` already produces, and exposes them as plain attributes:

```
import urllib.parse

class Request:
    def __init__(self, environ, params=None):
        self.environ = environ
        self.method = environ.get('REQUEST_METHOD', 'GET').upper()
        self.path = environ.get('PATH_INFO', '/')
        self.params = params or {}
        self._query_string = environ.get('QUERY_STRING', '')
        self._query = None
        self._body = None
```

`query` and `body` are both real Python properties, computed once on first access and cached — a genuine, deliberate design decision, not an arbitrary one, for reasons this chapter's own middle section verifies directly.

```
@property
def query(self):
    if self._query is None:
        parsed = urllib.parse.parse_qs(self._query_string)
```

```
self._query = {k: v[0] for k, v in parsed.items()}
return self._query
```

VERIFIED DIRECTLY — A REAL QUERY STRING, PARSED INTO A PLAIN DICT

A live request to `/users/42?active=true`, routed through Chapter 2's own `Router`, reaches a handler whose `req.query` returns `{'active': 'true'}` — `urllib.parse.parse_qs`, part of Python's own standard library, doing the real work; this class only wraps it lazily so a route that never reads `req.query` never pays the parsing cost at all.

Headers: A Real, Documented Gap in the WSGI Spec

Per PEP 3333, every real HTTP request header a client sends arrives in `environ` with an `HTTP_` prefix and underscores in place of hyphens — `X-API-Key` becomes the key `HTTP_X_API_KEY`. A `headers` property can reverse that mechanically:

```
@property
def headers(self):
    result = {}
    for key, value in self.environ.items():
        if key.startswith('HTTP_'):
            result[key[5:].replace('_', '-').title()] = value
    return result
```

VERIFIED DIRECTLY — CONTENT-TYPE NEVER SHOWS UP IN THIS DICT, ON PURPOSE

A real request sent with both a custom `X-API-Key` header and a real `Content-Type: application/json` header produces `req.headers` equal to `{'X-API-Key': 'secret-123', 'Host': ..., 'User-Agent': ..., 'Accept-Encoding': ..., 'Connection': ...}` — genuinely missing `Content-Type` entirely, even though it was really sent. This isn't a bug in the property above: PEP 3333 itself deliberately excludes `CONTENT_TYPE` and `CONTENT_LENGTH` from the `HTTP_`-prefixed set, storing them as their own separate, unprefixed `environ` keys instead. A `headers` property that only scans for `HTTP_*` keys will silently miss both, every time, on every real WSGI server — not just this one.

Reading `CONTENT_TYPE` back out correctly just means asking for the unprefixed key directly, exactly as this chapter's own `content_length` property (below) already does for its own equally

unprefixed sibling.

The Request Body: A Live Socket Is Not a BytesIO Object

`content_length` reads the same real, unprefixed key, guarding against the genuine case where it's missing entirely — a plain `GET` request with no body sends no `CONTENT_LENGTH` at all, and `int('')` raises a real `ValueError` if that's not handled:

```
@property
def content_length(self):
    raw = self.environ.get('CONTENT_LENGTH', '')
    try:
        return int(raw)
    except ValueError:
        return 0

@property
def body(self):
    if self._body is None:
        self._body = self.environ['wsgi.input'].read(self.content_length)
    return self._body
```

VERIFIED DIRECTLY — A REAL GET REQUEST, CONTENT_LENGTH GENUINELY ABSENT

A real live `GET` request through this exact code returns `content_length=0 body=b''` — no crash, no exception, despite there being no `CONTENT_LENGTH` key in `environ` for it at all.

`body` reads exactly `content_length` bytes from `environ['wsgi.input']` — the real request-body stream PEP 3333 itself warns about directly: *"the application should not attempt to read more data than is specified by the CONTENT_LENGTH variable,"* and *"the server is not required to read past the client's specified Content-Length."* Reading a bounded, correct amount looks safe. Whether it's actually safe to read *twice* turns out to depend entirely on what kind of stream is behind `wsgi.input` — verified two genuinely different ways below.

```
import io

# a fake, in-memory environ -- the kind a quick unit test might use
fake_stream = io.BytesIO(b'name=test')
```

```
print(fake_stream.read(9))    # b'name=test'
print(fake_stream.read(9))    # b'' -- looks perfectly safe
```

VERIFIED DIRECTLY — A SECOND READ ON BYTESIO JUST RETURNS EMPTY, INSTANTLY

Reading past the end of an in-memory `BytesIO` object behaves exactly the way most Python programmers already expect a stream to behave: it hits a real, immediate end-of-file and returns `b''`. A unit test built on a hand-constructed `environ` dict like this one would show `Request.body` accessed twice returning the identical cached bytes both times, with nothing to suggest any danger exists.

A real, live `wsgi.input` — the one a genuine `wsgiref` server actually hands a handler — is backed by a live TCP socket, not a bounded in-memory buffer. Three scenarios, run against a real server and a real client, each timed:

SCENARIO	WHAT IT DOES	REAL, MEASURED RESULT
<code>single_cached</code>	One handler reads <code>req.body</code> exactly once	200 OK in 0.02s
<code>double_cached</code>	One handler reads <code>req.body</code> twice; caching means the stream is only ever touched by the first access	200 OK in 0.03s — both reads return the same cached object (<code>first is second</code> → <code>True</code>)
<code>raw_peek_then_cached</code>	A handler reads <code>environ['wsgi.input']</code> directly first (a correctly-bounded read, using the real Content-Length), then constructs a <code>Request</code> and reads <code>req.body</code> afterward	Client times out — verified hanging past a forced 3-second cutoff, with the request never completing

VERIFIED DIRECTLY — A CORRECTLY-BOUNDED SECOND READ STILL HANGS FOREVER ON A REAL CONNECTION

The dangerous scenario above isn't calling `.read()` with no size argument at all — it's calling a properly bounded `.read(content_length)` a *second* time, after the real body bytes have already been consumed once. On a real, persistent HTTP connection, the client has finished sending exactly those bytes and is now waiting for a response; there is no more data coming, and a live socket has no built-in concept of "logically done with the HTTP body" the way a bounded `BytesIO` object does — it just keeps waiting for more bytes to arrive. The very first version of this experiment, using a genuinely unbounded `.read()` with no argument at all, was left running for over two real minutes with no response before it had to be forcibly terminated.

THIS IS EXACTLY WHY REQUEST.BODY CACHES, AND WHY IT MATTERS MORE THAN IT LOOKS

`Request.body`'s own `if self._body is None:` check is the entire reason `double_cached` above completes in 0.03 seconds instead of hanging: the underlying stream is read from `environ['wsgi.input']` **exactly once**, no matter how many times any code later asks for `request.body`. Any code path — a handler and a piece of middleware both wanting to inspect the body, for instance, a real scenario this course's own Chapter 5 builds toward — that reaches the raw stream a second time, bypassing this cache, reintroduces the exact hang verified above. And because an in-memory `BytesIO`-backed test can't reproduce this at all, a test suite built entirely on fake environs can pass cleanly while this exact bug ships to a real server.

Form Data: Reusing body()

A URL-encoded `POST` body is parsed the same way a query string is — `form()` reuses `body` (and therefore its caching) directly:

```
def form(self):
    parsed = urllib.parse.parse_qs(self.body.decode('utf-8'))
    return {k: v[0] for k, v in parsed.items()}
```

VERIFIED DIRECTLY AGAINST A REAL POST REQUEST

A real request sending `name=Ada&age=30` as its body reaches a handler whose `req.form()` returns exactly `{'name': 'Ada', 'age': '30'}`.

Building the Response Class

Chapter 2's own `app()` built every status line and header list by hand, in every branch. `Response` moves that into one place, and computes `Content-Length` automatically — a handler never has to get it right or wrong:

```
import json

class Response:
    STATUS_MESSAGES = {
        200: 'OK', 201: 'Created', 204: 'No Content',
```

```

301: 'Moved Permanently', 302: 'Found',
400: 'Bad Request', 404: 'Not Found', 405: 'Method Not Allowed',
500: 'Internal Server Error',
}

def __init__(self, body='', status=200, headers=None, content_type='text/plain'):
    self.body = body
    self.status = status
    self.headers = list(headers) if headers else []
    self.content_type = content_type

def status_line(self):
    return f"{self.status} {self.STATUS_MESSAGES.get(self.status, 'Unknown')}"

def to_wsgi(self, start_response):
    body_bytes = self.body.encode('utf-8') if isinstance(self.body, str) else self.body
    header_list = [
        ('Content-type', self.content_type),
        ('Content-Length', str(len(body_bytes))),
    ] + self.headers
    start_response(self.status_line(), header_list)
    return [body_bytes]

```

VERIFIED DIRECTLY — CORRECT CONTENT-LENGTH, EVERY TIME, WITH ZERO MANUAL WORK

A handler that returns `Response('a handler never has to compute Content-Length by hand')` and nothing else produces a real response whose measured body length and whose `Content-Length` header both come back as `53` — identical, verified from the client side, without the handler itself ever mentioning the number 53 anywhere.

Two Real Convenience Constructors: `redirect()` and `json()`

```

@classmethod
def redirect(cls, location, status=302):
    return cls(body='', status=status, headers=[('Location', location)])

@classmethod
def json(cls, data, status=200):
    return cls(body=json.dumps(data), status=status, content_type='application/json')

```

VERIFIED DIRECTLY — BOTH REAL REDIRECT BEHAVIORS, AND A REAL JSON BODY

`Response.redirect('/new-page')` produces a real `302 Found` with a real `Location: /new-page` header — confirmed two ways: `urllib`'s own default redirect-following behavior transparently follows it (landing on a genuine 404 there, since `/new-page` was never actually registered in this demo — a real, honest confirmation that the redirect is genuinely followed end to end, not just returned), and a second request made through a custom opener with redirect-following disabled reads the raw `302` status and the exact `/new-page` Location header directly. `Response.json({'id': 7, 'name': 'Ada'})` produces the real body `b'{"id": 7, "name": "Ada"}'` with `Content-type: application/json`, verified byte for byte.

Wiring Request & Response Into the Router

`app()` shrinks to three real cases — match, 405, 404 — and every handler below returns a `Response` instead of touching `start_response`:

```
router = Router()
router.add_route('GET', '/', lambda req: Response('Home page'), name='home')
router.add_route('GET', '/users/<int:id>',
                 lambda req: Response(f"User {req.params['id']}, query={req.query}")), name='users'
router.add_route('POST', '/submit', lambda req: Response(f"Got form: {req.form()}")), name='submit'
router.add_route('GET', '/old-page', lambda req: Response.redirect('/new-page'), name='old-page')
router.add_route('GET', '/api/users/<int:id>',
                 lambda req: Response.json({'id': req.params['id'], 'name': 'Ada'})), name='api-users'

def app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ.get('PATH_INFO', '/')
    status, handler, params, allowed = router.resolve(method, path)

    if status == 'MATCH':
        request = Request(environ, params)
        response = handler(request)
        return response.to_wsgi(start_response)
    elif status == '405':
        response = Response('405 Method Not Allowed', status=405,
                           headers=[('Allow', ', '.join(sorted(allowed)))])
        return response.to_wsgi(start_response)
    else:
```

```
response = Response('404 Not Found', status=404)
return response.to_wsgi(start_response)
```

Served through the real `wsgiref` server and hit with genuine `urllib.request` calls:

```
GET / → 200 b'Home page'
GET /users/42?active=true → 200 b"User 42, query={'active': 'true'}"
POST /submit → 200 b"Got form: {'name': 'Ada', 'age': '30'}"
GET /api/users/7 → 200 b'{"id": 7, "name": "Ada"}'
```

NOTHING IN CHAPTER 2'S OWN ROUTER CHANGED AT ALL

`Router.resolve()` is completely unaware that its own third return value is about to become `request.params`, or that its `handler` is now expected to return a `Response` instead of a plain string. This is deliberate — Chapter 2's own class is a real, stable dependency this chapter builds on top of, not something that had to change to support it.

Where This Course Is Headed

Chapter 4 builds a real template engine — parsing, compiling, rendering — so a handler can return `Response(render('page.html', title='Home'))` instead of hand-assembling an HTML string with f-strings.

Hands-On Exercises

EXERCISE 1

Add a `content_type` property to this chapter's own `Request` class that reads the real, unprefixed `CONTENT_TYPE` key directly from `environ`, and verify it against a real live request that sends both a custom header and a real `Content-Type` header — confirming `content_type` correctly returns the real value while `headers` (the `HTTP_*`-only property) still doesn't contain it.

 [View solution](#)

EXERCISE 2

Using a real, live wsgiref server (not an in-memory BytesIO-backed environ), write a handler that reads `environ['wsgi.input']` directly with a correctly-bounded read before ever constructing a `Request`, then calls `request.body` afterward — using a short client-side timeout (e.g. 3 seconds) to confirm the request hangs rather than waiting indefinitely — and explain concretely why the identical scenario tested against a BytesIO-backed fake environ would NOT reveal this bug at all.

 [View solution](#)
EXERCISE 3

Add a `Response.html()` classmethod that behaves like `Response.json()` but sets `content_type='text/html'` instead, register a route returning `Response.html('<h1>Hello</h1>')`, and verify the real Content-type header a live client receives is exactly `text/html` rather than the default `text/plain`.

 [View solution](#)
CHAPTER 3 QUICK REFERENCE

- **Request** — wraps environ + Router's own params dict; method, path, params as plain attributes
- **query** — a lazy, cached property parsing QUERY_STRING via `urllib.parse.parse_qs`
- **headers** — built from HTTP_* environ keys only; verified to genuinely exclude Content-Type/Content-Length, which PEP 3333 deliberately stores unprefixd
- **content_length / body** — `content_length` safely defaults to 0 on a real missing-header GET request; `body` caches its single real read from `wsgi.input`
- **The double-read danger** — verified directly: an in-memory BytesIO-backed test returns `b''` instantly on a second read and looks completely safe; the identical code against a real live socket-backed server hangs indefinitely instead, confirmed with a forced timeout
- **Response** — `status_line()/to_wsgi()` compute Content-Length automatically; verified matching the real body length exactly, every time
- **redirect() / json()** — two real convenience constructors, verified against genuine 302/Location and application/json responses

- **Next chapter:** A real template engine — parsing, compiling & rendering

CHAPTER 4: BUILDING A TEMPLATE ENGINE: PARSING, COMPILING & RENDERING

Building a Web Framework

Chapter 4 · Building a Template Engine: Parsing, Compiling & Rendering

Chapter 3 closed on a real handler shape this course hadn't built yet:

`Response(render('page.html', title='Home'))`. Every handler written so far builds its own HTML with an f-string, which works for one line and falls apart the moment a real page needs a shared layout, a loop over real data, or a value nobody has separately checked for safety. This chapter builds that missing piece — a real, file-based `TemplateEngine` class this framework keeps and reuses from here on — and runs directly into the one design decision a template engine can't get wrong: what happens to a value the instant it lands inside real HTML.

The Basic Job: Substituting Into a Real Template File

A template engine's simplest possible form finds placeholders in a string and replaces them with real values — the same regex-substitution shape Chapter 2's own router used for dynamic path segments, applied to a whole document loaded from disk instead of a URL:

templates/basic.html

```
Hello, {{ name }}!
```

```
import re

def render_basic(path, context):
    with open(path, 'r', encoding='utf-8') as f:
        template = f.read()
    return re.sub(
        r'\{\{s*(\w+)\s*\}\}',
        lambda m: str(context.get(m.group(1), '')),
        template,
    )

print(render_basic('templates/basic.html', {'name': 'World'}))
# Hello, World!
```

Genuinely working, genuinely unsafe the moment a real, user-controlled value reaches it.

The Real Security Problem: Naive Interpolation

```
malicious = {'name': '<script>alert(1)</script>'}
print(render_basic('templates/basic.html', malicious))
# Hello, <script>alert(1)</script>!
```

VERIFIED DIRECTLY — A REAL, LIVE <SCRIPT> TAG REACHES THE OUTPUT INTACT

Loaded straight from a real `.html` file on disk and rendered with a real malicious context value, the output contains a genuine, executable `<script>` tag — not the text describing one. A profile display name, a comment, a search query echoed back — anything reaching `render_basic()` unescaped becomes a real, working cross-site scripting vector the instant it's rendered into an actual page.

Auto-Escaping by Default, With an Explicit Opt-Out

Every value gets HTML-escaped automatically unless it's deliberately marked as already safe — a plain `str` subclass is enough to carry that marker:

```
import html

class SafeString(str):
    # its only job is to be recognizable as "already safe" -- nothing else changes
    pass

def escape(value):
    if isinstance(value, SafeString):
        return str(value)
    return html.escape(str(value))
```

VERIFIED DIRECTLY — IDENTICAL TEXT, TWO GENUINELY DIFFERENT OUTCOMES

Rendering the exact same malicious string through `escape()` unwrapped returns `<script>alert(1)</script>` — inert, visible text. Wrapping the identical characters as `SafeString(...)` first returns them completely untouched. Safety isn't a property of the value itself;

it's a decision made once, deliberately, exactly at the one point a developer genuinely knows a value is already safe — never trusted implicitly by default.

A Real Node Tree, With Nested if/for

Real templates need loops and conditionals, and both can nest inside each other — an `if` inside a `for`, say. That needs a real parsed structure, not one flat regex pass. Four small node classes, each responsible only for rendering itself:

```
class TextNode:
    def __init__(self, text):
        self.text = text
    def render(self, ctx):
        return self.text

class VarNode:
    def __init__(self, name):
        self.name = name
    def render(self, ctx):
        return escape(ctx.get(self.name, ''))

class IfNode:
    def __init__(self, cond_name, body_nodes):
        self.cond_name = cond_name
        self.body_nodes = body_nodes
    def render(self, ctx):
        if ctx.get(self.cond_name):
            return ''.join(node.render(ctx) for node in self.body_nodes)
        return ''

class ForNode:
    def __init__(self, var, coll_name, body_nodes):
        self.var = var
        self.coll_name = coll_name
        self.body_nodes = body_nodes
    def render(self, ctx):
        out = []
        for item in ctx.get(self.coll_name, []):
            local = dict(ctx)
            local[self.var] = item
            for node in self.body_nodes:
```

```

        out.append(node.render(local))
    return ''.join(out)

```

A tokenizer splits the raw source on `{{ ... }}` and `{% ... %}` boundaries; a small recursive-descent parser walks the tokens once, recursing into a fresh call for each `if`/`for` body and returning control the moment it hits the matching `endif`/`endfor`:

```

TOKEN_RE = re.compile(r'(\{\{.*?\}\}|\{%.*?%\})', re.S)

def tokenize(template):
    return [t for t in TOKEN_RE.split(template) if t]

def parse(tokens, index=0):
    nodes = []
    while index < len(tokens):
        token = tokens[index]
        if token.startswith('{{'):
            nodes.append(VarNode(token[2:-2].strip()))
            index += 1
        elif token.startswith('%'):
            tag = token[2:-2].strip()
            if tag.startswith('if '):
                cond_name = tag[3:].strip()
                body, index = parse(tokens, index + 1)
                nodes.append(IfNode(cond_name, body))
            elif tag.startswith('for '):
                m = re.match(r'for (\w+) in (\w+)', tag)
                var, coll_name = m.group(1), m.group(2)
                body, index = parse(tokens, index + 1)
                nodes.append(ForNode(var, coll_name, body))
            elif tag in ('endif', 'endfor'):
                return nodes, index + 1
            else:
                index += 1
        else:
            nodes.append(TextNode(token))
            index += 1
    return nodes, index

def compile_template(source):
    nodes, _ = parse(tokenize(source))
    return nodes

```

VERIFIED DIRECTLY — A REAL NESTED IF-INSIDE-FOR, FROM A REAL FILE

A template of `{% for item in items %}{% if item %}<li>{{ item }}</li>{% endif %}{% endfor %}`, compiled once and rendered against `{'items': ['a', '', 'b', None, 'c']}`, produces exactly `<li>a</li><li>b</li><li>c</li>` — the two falsy items silently skipped, with the nested `IfNode` correctly evaluated fresh against each loop iteration's own local context.

Template Inheritance: extends & block, From Real Files

A shared layout lives in one parent file; a child file `extends` it and overrides only the regions it names:

templates/base.html

```
<html><head><title>{% block title %}My Site{% endblock %}</title></head><body>{% block
```

templates/child.html

```
{% extends "base.html" %}
{% block content %}Welcome, {{ user }}!{% endblock %}
```

`child.html` deliberately never mentions `title` at all — only `content`.

Resolving a child means reading both real files, extracting the child's own named blocks with one regex, and substituting each into the parent's matching block — before the merged result ever reaches `compile_template()`, so anything inside a block still gets the full real `{{ }}`/`{% %}` treatment:

```
def extract_blocks(text):
    blocks = {}
    for m in re.finditer(r'\{%s*block\s+(\w+)\s*%\}(.*)\{%s*endblock\s*%\}', text, re.DOTALL):
        blocks[m.group(1)] = m.group(2)
    return blocks

def substitute_blocks(parent_text, child_blocks):
    def replace(m):
        name, content = m.group(1), child_blocks.get(name, '')
        return f'{{{{ content }}}}'
```

```

        return child_blocks.get(m.group(1), m.group(2))    # child wins; else keep parent's
    return re.sub(r'\{%s*block\s+(\w+)\s*%\}(.*)\{%s*endblock\s*%\}', replace, parent_raw)

```

Chaining the pieces manually, exactly the way a class will soon do internally:

```

def read_file(path):
    with open(path, 'r', encoding='utf-8') as f:
        return f.read()

child_raw = read_file('templates/child.html')
parent_raw = read_file('templates/base.html')
merged = substitute_blocks(parent_raw, extract_blocks(child_raw))
nodes = compile_template(merged)
print(''.join(n.render({'user': 'Ada'}) for n in nodes))
# <html><head><title>My Site</title></head><body>Welcome, Ada!</body></html>

```

VERIFIED DIRECTLY — TITLE FALLS BACK TO THE PARENT'S OWN REAL DEFAULT

`child.html` only ever defines `content`; its real, rendered output nonetheless shows `<title>My Site</title>`, taken straight from `base.html`'s own default text, since `extract_blocks()` never found a `title` block in the child at all, and `substitute_blocks()`'s own `.get(m.group(1), m.group(2))` falls back to the parent's original content the instant a key is genuinely missing. A child only has to say what's genuinely different about it.

Building the Real TemplateEngine Class

Everything above — reading a real file, resolving `extends`, compiling to a node tree — becomes one class that owns a template directory and caches the compiled result, parsed once per template name and reused on every later render, exactly the discipline Web Framework Internals' own Chapter 4 verified Django's and Jinja2's real engines both follow:

```

import os

class TemplateEngine:
    def __init__(self, template_dir):
        self.template_dir = template_dir
        self._cache = {}

```

```

def _read_file(self, name):
    path = os.path.join(self.template_dir, name)
    with open(path, 'r', encoding='utf-8') as f:
        return f.read()

def _resolve_source(self, template_name):
    raw = self._read_file(template_name)
    m = re.match(r'\s*\{%s*extends\s+"([^\"]+)"\s*%\}', raw)
    if m:
        parent_raw = self._read_file(m.group(1))
        return substitute_blocks(parent_raw, extract_blocks(raw))
    return raw

def get_template(self, template_name):
    if template_name not in self._cache:
        self._cache[template_name] = compile_template(self._resolve_source(template_name))
    return self._cache[template_name]

def render(self, template_name, **context):
    nodes = self.get_template(template_name)
    return ''.join(node.render(context) for node in nodes)

```

A REAL, GENUINE BUG FOUND WHILE WRITING THIS EXACT CLASS

The first version of `render()` named its own template-name parameter `name`, matching Chapter 2's own `Router.add_route(..., name=...)` convention. It broke immediately, on the very first real call: `engine.render('basic.html', name='World')` raised `TypeError: render() got multiple values for argument 'name'` — `'basic.html'` filled the positional `name` parameter, and the keyword

`name='World'` (a completely ordinary, plausible context value) collided with it directly. Renaming the parameter to `template_name` fixed it outright. This is the exact reason Flask's own real `render_template()` never calls its own first parameter `name` either — `name` is too common a real context key for a template engine's own API to claim for itself.

VERIFIED DIRECTLY — COMPILE-ONCE CACHING ADDS ZERO EXTRA FILE READS

Instrumenting `_read_file()` to count real calls, then rendering the same already-cached template five times in a row, adds exactly **zero** further file reads. The file is opened and parsed on the very first render only; every render after that walks the same in-memory node tree Chapter 4's own compiler already built.

A Real, Live-Verified Limitation: Stale Caching

Caching by name has an honest cost: once a template is compiled, editing its file on disk mid-run changes nothing a running server actually serves.

```
# a real, live server, already running against templates/greeting.html:
# "<p>Hello, {{ name }}! (v1)</p>"

first_response = get('/') # <p>Hello, Ada! (v1)</p>

# the real file on disk is edited WHILE the server keeps running --
# greeting.html now reads "<p>Hi there, {{ name }}! (v2, updated on disk)</p>"

second_response = get('/') # <p>Hello, Ada! (v1)</p> -- still the OLD content
```

VERIFIED DIRECTLY AGAINST A REAL, LIVE RUNNING SERVER

Two genuine `GET` requests, sent to the same live `wsgiref` server, with the real template file edited on disk in between, both return the identical `v1` output — confirmed by actually running the edit and the second request against a process that never restarted. This isn't a bug so much as an unavoidable consequence of caching correctly by name: nothing in `TemplateEngine` has any way of knowing the file underneath a cached name has changed. Fixing this — detecting a changed file and recompiling automatically — is exactly the real job of the live-reload development server this course builds in Chapter 8.

Wiring TemplateEngine Into Response

A handler now renders a real file, through the real engine, and wraps the result in a `Response` exactly the way Chapter 3's own closing line promised. A third, separate real template exercises everything at once — extending the same `base.html`, but this time overriding *both* of its blocks:

templates/users.html

```
{% extends "base.html" %}
{% block title %}Users{% endblock %}
{% block content %}
<h1>Users</h1>
{% if empty %}<p>No users found.</p>{% endif %}
```

```
<ul>{% for user in users %}<li>{{ user }}</li>{% endfor %}</ul>
{% endblock %}
```

```
templates = TemplateEngine('templates')

router.add_route('GET', '/users', lambda req: Response(templates.render(
    'users.html', users=['Ada', 'Grace', '<b>Rogue</b>'], empty=False,
)))
router.add_route('GET', '/users/empty', lambda req: Response(templates.render(
    'users.html', users=[], empty=True,
)))
```

Served through the real `wsgiref` app and hit with genuine `urllib.request` calls:

```
GET /users
→ <html><head><title>Users</title></head><body><h1>Users</h1>
  <ul><li>Ada</li><li>Grace</li><li>&lt;b&gt;Rogue&lt;/b&gt;</li></ul></body></html>

GET /users/empty
→ <html><head><title>Users</title></head><body><h1>Users</h1>
  <p>No users found.</p><ul></ul></body></html>
```

VERIFIED DIRECTLY — EXTENDS, BLOCK, IF, FOR, AND ESCAPING, ALL CORRECT TOGETHER

Every mechanism this chapter built is exercised in one real response: the layout comes from `base.html`, the title and content blocks come from `users.html`, the empty-state `{% if %}` correctly fires only on the second route, the `{% for %}` loop correctly renders three real list items, and the third item's own literal `<b>Rogue</b>` string comes back fully escaped — confirming auto-escaping survives the full route-to-template pipeline, not just the isolated examples earlier in this chapter.

Where This Course Is Headed

Chapter 5 builds a real middleware pipeline — an onion-model chain wrapping every request and response, the mechanism a real framework would use to add authentication, logging, or CSRF protection around every route this chapter's own `Router` already dispatches to.

Hands-On Exercises

EXERCISE 1

Extend this chapter's own `IfNode` and `parse()` function to support a real `{% else %}` branch — `{% if flag %}...{% else %}...{% endif %}` — and verify it against both a truthy and a falsy value for `flag`, plus a genuinely nested case with an `if/else` living inside a `for` loop.

 [View solution](#)

EXERCISE 2

Write a real `render_bio_markdown(raw_text)` function that escapes its input completely first, then converts `**bold**` markers on the now-safe text into real `<strong>` tags, returning the result as a `SafeString` — and verify against a single real input containing both a `**bold**` marker and a literal `<script>` tag that only the bold marker becomes real markup while the script tag stays fully inert.

 [View solution](#)

EXERCISE 3

Reproduce this chapter's own stale-cache finding yourself against a real, live `wsgiref` server serving a route rendered through `TemplateEngine`: make one real request, edit the template file on disk while the server keeps running, make a second real request, and confirm both responses are identical — then explain in your own words why `get_template()`'s own caching is what causes this, not a bug in `compile_template()` itself.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **The basic job** — read a real `.html` file, substitute `{{ }}` placeholders from a context dict
- **The real security problem** — unescaped interpolation puts a real, live `<script>` tag straight into rendered output
- **Auto-escaping by default** — `escape()` escapes everything except values explicitly wrapped as `SafeString`

- **The node tree** — TextNode/VarNode/IfNode/ForNode, parsed once via a real recursive-descent tokenizer/parser supporting nested if/for
- **extends/block inheritance** — child blocks substituted into the parent's raw text before compilation, verified falling back to the parent's own default for any block a child leaves untouched
- **TemplateEngine** — real file loading, inheritance resolution, and compile-once caching in one class; a genuine name/template_name naming-collision bug found and fixed along the way
- **Stale caching** — verified live: editing a cached template's file on disk doesn't change what a running server serves, until Chapter 8's own live-reload dev server fixes it
- **Next chapter:** A real onion-model middleware pipeline

CHAPTER 5: MIDDLEWARE: A REAL ONION-MODEL PIPELINE

Building a Web Framework

Chapter 5 · Middleware: A Real Onion-Model Pipeline

Every chapter so far has hand-wired its own `app(envIRON, start_response)` function, manually calling `Router.resolve()` and manually turning the result into a `Response`. That works for one route file, but it has no place to put code that should run around *every* request — authentication, logging, timing, error handling — without editing every single handler to add it. This chapter builds that missing layer: a real middleware pipeline, wired into a new, persistent `App` class that finally becomes the actual WSGI callable Chapter 1 first introduced, replacing every hand-rolled `app()` function this course has written since.

The Basic Job: A Chain of Functions Wrapping "Next"

A middleware is a function taking the request and a reference to whatever comes next — another middleware, or the real handler at the center. Composing a list means nesting each one inside the next:

```
def build_chain(middlewares, handler):
    chain = handler
    for middleware in reversed(middlewares):
        chain = (lambda mw, nxt: lambda request: mw(request, nxt))(middleware, chain)
    return chain

def middleware_a(request, next_):
    trace.append('before A')
    response = next_(request)
    trace.append('after A')
    return response

# middleware_b, middleware_c follow the identical shape

def toy_handler(request):
    trace.append('handler')
    return Response('OK')

chain = build_chain([middleware_a, middleware_b, middleware_c], toy_handler)
```

```
chain(Request({'REQUEST_METHOD': 'GET', 'PATH_INFO': '/'}))
print(trace)
```

VERIFIED DIRECTLY — REAL, GENUINELY NESTED EXECUTION, NOT SEQUENTIAL

`trace` comes out as `['before A', 'before B', 'before C', 'handler', 'after C', 'after B', 'after A']`. Each middleware's own "after" code runs in the exact reverse of registration order — A registers first and finishes last, since A's own call to `next_(request)` is what invokes everything nested inside it. Real frameworks name this precisely — Django's own documentation: "You can think of it like an onion: each middleware class is a 'layer' that wraps the view, which is in the core of the onion."

Building the Real App Class: A Router, a Middleware List, and a WSGI Callable

Every piece from Chapters 2–4 — `Router`, `Request`, `Response` — folds into one persistent class that owns a middleware list too, and implements `__call__(environ, start_response)` directly, the exact real interface Chapter 1 verified against `wsgiref`'s own reference server:

```
class App:
    def __init__(self):
        self.router = Router()
        self.middlewares = []
        self._chain = None

    def add_route(self, method, path, handler, name=None):
        self.router.add_route(method, path, handler, name=name)
        self._chain = None # a new route can change what the innermost handler does

    def use(self, middleware):
        self.middlewares.append(middleware)
        self._chain = None # force a rebuild on the next real request

    def _dispatch(self, request):
        status, handler, params, allowed = self.router.resolve(request.method, request.path)
        if status == 'MATCH':
            request.params = params
            return handler(request)
        elif status == '405':
```

```

        return Response('405 Method Not Allowed', status=405,
                        headers=[('Allow', ', '.join(sorted(allowed))))])
    else:
        return Response('404 Not Found', status=404)

    def _build_chain(self):
        chain = self._dispatch # the innermost "handler" IS route dispatch itself
        for mw in reversed(self.middlewares):
            chain = (lambda m, nxt: lambda req: m(req, nxt))(mw, chain)
        return chain

    def __call__(self, environ, start_response):
        if self._chain is None:
            self._chain = self._build_chain()
        request = Request(environ)
        response = self._chain(request)
        return response.to_wsgi(start_response)

```

The middleware chain is now built once, on first use, and cached — the exact same "build once, reuse" discipline Chapter 4's own `TemplateEngine` already established, invalidated only when a new route or middleware is actually registered.

VERIFIED DIRECTLY — AN APP INSTANCE IS A REAL WSGI APPLICATION, WITH NOTHING WRAPPED AROUND IT

`make_server('127.0.0.1', 8861, app)` — passing the `App` instance itself, not a function — served a real request correctly. Python's own real `wsgiref` server only ever cared that its third argument is callable with `(environ, start_response)`; a class instance with a real `__call__` method satisfies that identically to the plain functions every prior chapter used. This is the exact same real interface Chapter 1 verified against PEP 3333, now implemented by a genuine, stateful object instead of a bare function.

Short-Circuiting: A Real Authentication Middleware

to — an unauthenticated request has no business reaching the real handler:

```

def auth_middleware(request, next_):
    if request.environ.get('HTTP_X_API_KEY') != 'secret123':
        return Response('401 Unauthorized', status=401) # next_() is never called

```

```
return next_(request)
```

```
app = App()
app.use(auth_middleware)
app.add_route('GET', '/secret', lambda req: Response('secret data'))
```

VERIFIED DIRECTLY AGAINST A REAL, LIVE SERVER

Three genuine requests, sent to the same running `App`: no `X-API-Key` header returns a real `401`; a wrong key also returns `401`; the correct key, `secret123`, returns `200` and the real handler's own body. The rejected requests never reach the handler at all — verified by a second, chained `logging_middleware` that never records anything for either of the two rejected requests.

The Real Cost of Forgetting to Call `next_()`

Short-circuiting on purpose is a feature. Forgetting to call `next_()` at all, and not returning anything either, is a real, easy mistake — Express's own documentation names the toy-version consequence directly: "Otherwise, the request will be left hanging." A real WSGI app built on this chapter's own `App` class has a genuinely worse, more concrete version of that same mistake:

```
def broken_middleware(request, next_):
    print('ran, but forgot to call next_() or return anything')
    # no call to next_(request); no return statement at all

app = App()
app.use(broken_middleware)
app.add_route('GET', '/', lambda req: Response('should never be reached'))
```

VERIFIED DIRECTLY — A REAL CRASH, NOT A SILENT NONE

Hit with a genuine live request, the server's own console shows a real Python traceback:

`AttributeError: 'NoneType' object has no attribute 'to_wsgi'`, raised from `App.__call__`'s own `response.to_wsgi(start_response)` line — `self._chain(request)` genuinely returned `None`, and nothing downstream of that was ever written to be `None`-safe. `wsgiref` itself catches that exception and produces a real HTTP `500`, with a body of exactly `"A server error occurred. Please`

`contact the administrator."` — the client receives a completely real, working 500 response, just one carrying zero information about what actually broke.

Mutating the Request and the Response

Code running *before* `next_()` can attach data every downstream layer can read — this chapter's own `Request` class gets a plain `state` dict specifically for this:

```
def inject_user_middleware(request, next_):
    request.state['user'] = {'id': 42, 'name': 'Sam'}
    return next_(request)
```

Code running *after* `next_()` returns can inspect or modify the real response on its way back out — the exact real example FastAPI's own documentation uses to introduce middleware at all is a timing header:

FASTAPI'S OWN DOCUMENTATION, QUOTED DIRECTLY

"You can add code to be run with the `request`, before any path operation receives it. And also after the `response` is generated, before returning it." FastAPI's own real example measures elapsed time around `call_next(request)` and sets `response.headers["X-Process-Time"]` with the result.

```
def timing_middleware(request, next_):
    start = time.perf_counter()
    response = next_(request)
    elapsed = time.perf_counter() - start
    response.headers.append(('X-Process-Time', f"{elapsed:.6f}"))
    return response

def slow_handler(request):
    time.sleep(0.05)
    user = request.state.get('user')
    return Response(f"Hello, {user['name']} (id={user['id']})")

app = App()
app.use(inject_user_middleware)
```

```
app.use(timing_middleware)
app.add_route('GET', '/', slow_handler)
```

VERIFIED DIRECTLY — A REAL, ACCURATELY-MEASURED ELAPSED TIME FROM A REAL LIVE REQUEST

`slow_handler` genuinely sleeps for 0.05 real seconds; the live response body reads `Hello, Sam (id=42)` — confirming `inject_user_middleware`'s own write to `request.state` really was visible two layers downstream — and the real `X-Process-Time` header measured `0.050328`, accurate to a fraction of a millisecond of real overhead, independently reproducing FastAPI's own documented example on this framework's own code.

Error-Handling Middleware: With It, and Genuinely Without It

Since `next_()` is an ordinary function call, an exception raised deep in the chain propagates upward through every enclosing layer exactly like any other Python exception — a middleware can catch it with a plain `try` / `except` around that one call:

```
def error_handling_middleware(request, next_):
    try:
        return next_(request)
    except ValueError as e:
        return Response(f"500 Internal Server Error: {e}", status=500)

def crashing_handler(request):
    raise ValueError('something genuinely went wrong')
```

Registering the identical `crashing_handler` two ways — with the middleware, and without it — on two separate live servers:

CONFIGURATION	REAL, LIVE CLIENT RESULT
error_handling_middleware present	<code>500</code> , body exactly <code>500 Internal Server Error: something genuinely went wrong</code>
no error-handling middleware at all	<code>500</code> , body exactly <code>A server error occurred. Please contact the administrator.</code>

VERIFIED DIRECTLY, LIVE, BOTH CONFIGURATIONS — THE REAL DIFFERENCE IS WHAT THE CLIENT ACTUALLY LEARNS

Both configurations genuinely return a real `500` — the raw `ValueError`, left uncaught, propagates all the way out of `App.__call__` itself and is caught by `wsgiref`'s own internal handler exactly the same way Chapter 5's own `broken_middleware` case was, producing the identical generic message. With `error_handling_middleware` in place, the client instead receives this framework's own specific, real message — `"something genuinely went wrong"`, the exact real exception text, not a generic placeholder. Nothing here is special-cased for errors specifically; a real `try / except` around one ordinary function call is the entire mechanism, and skipping it doesn't lose the request — it loses the information a real developer or user would actually need.

Where This Course Is Headed

Chapter 6 builds real sessions and cookies — state that survives across genuinely separate, stateless HTTP requests — almost certainly implemented as a middleware itself, reading a session cookie *before* `next_()` and writing an updated one *after* it returns, using exactly the two mutation points this chapter just verified.

Hands-On Exercises

EXERCISE 1

Add a `cors_middleware` to this chapter's own `App` system that adds an `Access-Control-Allow-Origin: *` header to every response after `next_()` returns, register it on a real `App` with `app.use()`, and verify against a real live request that the header genuinely arrives on the client's own response.

 [View solution](#)

EXERCISE 2

Using this chapter's own `auth_middleware` and a new `logging_middleware` that appends to `request.state` before and after calling `next_()`, build two real `Apps` that register the two middlewares in opposite orders (`auth-then-logging` vs. `logging-then-auth`), send an unauthenticated real request to each, and verify from `request.state` whether `logging_middleware`'s own "before" code actually ran before the rejection in each case.

[View solution](#)

EXERCISE 3

Define a real `NotFoundError` exception class, extend this chapter's own `error_handling_middleware` to catch it separately from `ValueError` and return a distinct real 404 response for it, and verify live against two separate routes — one whose handler raises `ValueError`, one whose handler raises `NotFoundError` — that each produces its own distinct, correct real status code and body.

[View solution](#)

CHAPTER 5 QUICK REFERENCE

- **The basic job** — each middleware wraps "whatever comes next," composed into one nested call chain
- **The real onion order** — verified as before-A/before-B/before-C/handler/after-C/after-B/after-A, matching Django's own documented "onion" model
- **App** — a real, persistent class owning Router + a middleware list, implementing `__call__(environ, start_response)` directly; verified as a genuine WSGI app handed straight to `wsgiref`, no wrapper function needed
- **Short-circuiting** — a middleware can return a response without calling `next_()`, verified stopping every inner layer, including a chained logging middleware, from running at all
- **Forgetting to call `next_()`** — verified as a real crash in this chapter's own App, caught by `wsgiref` and surfaced as its own generic 500, not a silent `None`
- **Request/response mutation** — `request.state` before `next_()`; `response.headers` after `next_()` returns — verified with a real, measured `X-Process-Time` header
- **Error-handling middleware** — an ordinary `try/except` around one call to `next_()`; verified live that it's the difference between the client receiving this framework's own real error message and `wsgiref`'s own generic one
- **Next chapter:** Sessions and cookies — real state across stateless requests

CHAPTER 6: SESSIONS & COOKIES: REAL STATE ACROSS STATELESS REQUESTS

Building a Web Framework

Chapter 6 · Sessions & Cookies: Real State Across Stateless Requests

Every request this course has built so far starts from nothing — a fresh `environ` dict, a fresh `Request`, no memory of anything a previous request did. That's genuinely how HTTP works: each request really is independent. A shopping cart, a "logged in" state, a visit counter all need something to survive across that gap anyway — and this chapter builds it exactly the way Chapter 5's own closing paragraph predicted: as ordinary middleware, reading a cookie before `next_()` and writing one after it returns.

Cookies: The Real Header on Both Sides

A cookie is nothing more than a header. The server sends `Set-Cookie` in a response; the browser sends it straight back as `Cookie` on every later request to the same site. Python's own standard library already parses and builds the real, correct syntax —

`http.cookies.SimpleCookie`:

```
from http.cookies import SimpleCookie

c = SimpleCookie()
c['session_id'] = 'abc123'
c['session_id']['path'] = '/'
c['session_id']['max-age'] = 3600
c['session_id']['httponly'] = True
c['session_id']['samesite'] = 'Lax'
print(c['session_id'].OutputString())
# session_id=abc123; HttpOnly; Max-Age=3600; Path=/; SameSite=Lax

parser = SimpleCookie()
parser.load('session_id=abc123; theme=dark')
print({k: m.value for k, m in parser.items()})
# {'session_id': 'abc123', 'theme': 'dark'}
```

Wired into this chapter's own real `Request` and `Response` classes:

```

# added to Request (Chapter 3):
@property
def cookies(self):
    if self._cookies is None:
        jar = SimpleCookie()
        jar.load(self.environ.get('HTTP_COOKIE', ''))
        self._cookies = {k: m.value for k, m in jar.items()}
    return self._cookies

# added to Response (Chapter 3):
def set_cookie(self, name, value, max_age=None, path='/', http_only=True, same_site=None):
    cookie = SimpleCookie()
    cookie[name] = value
    cookie[name]['path'] = path
    if max_age is not None:
        cookie[name]['max-age'] = max_age
    if http_only:
        cookie[name]['httponly'] = True
    cookie[name]['samesite'] = same_site
    self.headers.append(('Set-Cookie', cookie[name].OutputString()))

```

A Real Session Store

Cookies alone can only carry text the client sends back verbatim — anything sensitive still needs to live on the server, keyed by an ID the cookie merely points at. A minimal in-memory store, with a genuine, per-session idle-timeout:

```

import secrets, time

class SessionStore:
    def __init__(self, max_age=3600):
        self._sessions = {} # sid -> (data_dict, created_at)
        self.max_age = max_age

    def create(self):
        sid = secrets.token_urlsafe(32)
        self._sessions[sid] = ({}, time.time())
        return sid

    def load(self, sid):

```

```

    entry = self._sessions.get(sid)
    if entry is None:
        return None
    data, created_at = entry
    if time.time() - created_at > self.max_age:
        del self._sessions[sid]
        return None
    return data

def save(self, sid, data):
    _, created_at = self._sessions.get(sid, ({}, time.time()))
    self._sessions[sid] = (data, created_at)

```

Wiring It in as Ordinary Middleware

No special-case machinery needed — the exact before-`next_()` /after-`next_()` shape Chapter 5 already built, reading the session cookie first and writing an updated one last:

```

def make_session_middleware(store, cookie_max_age=3600):
    def middleware(request, next_):
        sid = request.cookies.get('session_id')
        data = store.load(sid) if sid else None
        if data is None:
            sid = store.create()
            data = store.load(sid)
        request.state['session'] = data
        response = next_(request)
        store.save(sid, request.state['session'])
        response.set_cookie('session_id', sid, max_age=cookie_max_age)
        return response
    return middleware

def visit_counter_handler(request):
    session = request.state['session']
    session['visits'] = session.get('visits', 0) + 1
    return Response(f"You have visited {session['visits']} times")

store = SessionStore()
app = App()
app.use(make_session_middleware(store))
app.add_route('GET', '/', visit_counter_handler)

```

The Real Payoff: Why Sessions Need Cookies

Run the exact same code two genuinely different ways against a real live server:

```
# no cookie jar at all -- two plain, independent requests
urllib.request.urlopen('http://127.0.0.1:8891/').read() # b'You have visited 1 times'
urllib.request.urlopen('http://127.0.0.1:8891/').read() # b'You have visited 1 times'

# a real cookie jar, mimicking an actual browser
jar = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(jar))
opener.open('http://127.0.0.1:8891/').read() # b'You have visited 1 times'
opener.open('http://127.0.0.1:8891/').read() # b'You have visited 2 times'
opener.open('http://127.0.0.1:8891/').read() # b'You have visited 3 times'
```

VERIFIED DIRECTLY — IDENTICAL SERVER CODE, GENUINELY DIFFERENT REAL CLIENT BEHAVIOR

Two plain `urlopen()` calls, with no jar, each receive their own real, distinct `Set-Cookie` header with a completely different session ID — and because nothing ever sends either cookie back, `request.cookies.get('session_id')` finds nothing on the second call, so `make_session_middleware` genuinely creates a brand-new session every single time. The count stays at `1` forever, not because of a bug, but because this client is never actually reusing anything. Switching to a real `http.cookiejar.CookieJar` — which automatically stores every real `Set-Cookie` it receives and resends it on every later request to the same host, exactly what a real browser does — makes the identical server-side code correctly count `1, 2, 3`. The session was never really "in" the server alone; it only works because the cookie genuinely round-trips.

Server-Side Expiry Is a Genuinely Separate Control From the Cookie's Own Max-Age

A cookie's own `Max-Age` only governs when a well-behaved client stops sending it — nothing stops a client from ignoring that and resending an old cookie anyway. Real security has to come from the server checking independently, which is exactly what `SessionStore.load()`'s own timeout already does. Isolating that from the cookie's client-side lifetime means deliberately setting the two to genuinely different real values:

```
store = SessionStore(max_age=1) # the SERVER's own 1-s
app.use(make_session_middleware(store, cookie_max_age=3600)) # the COOKIE's own much 1
```

VERIFIED DIRECTLY — THE CLIENT STILL HOLDS AND STILL SENDS THE EXACT SAME COOKIE; ONLY THE SERVER REJECTS IT

Real request 1 counts **1**; real request 2, immediately after, counts **2**. After sleeping 1.5 real seconds — past the server's own 1-second timeout, nowhere near the cookie's own 3600-second **Max-Age** — the jar was checked directly and confirmed still holding, and about to resend, the *exact same* session ID from request 1. The real third request nonetheless counts **1** again: a brand-new session, created because `SessionStore.load()` found the old one past its own real timeout and deleted it — entirely independent of whether the client ever intended to stop using that cookie.

Session ID Security: Unpredictability, and HttpOnly

A session ID that could be guessed would let an attacker impersonate another real user with zero access to their machine or cookies at all — which is exactly why `secrets.token_urlsafe()`, not `random` or a counter, generates them:

```
ids = set()
for _ in range(20000):
    ids.add(secrets.token_urlsafe(32))
print(len(ids))
# 20000 -- zero collisions
```

VERIFIED DIRECTLY — 20,000 REAL, INDEPENDENTLY GENERATED IDS, ZERO COLLISIONS

`secrets.token_urlsafe(32)` draws from Python's own real, documented cryptographically-secure random source, producing a 43-character real string with 256 bits of genuine entropy — 20,000 real calls in a row produced 20,000 real, distinct values.

`set_cookie()`'s own default of `http_only=True` ties directly back to Chapter 4's own real, verified XSS vulnerability — unescaped output there let a genuine `<script>` tag execute in the page. Without `HttpOnly`, that same script could read `document.cookie` and steal a real session ID outright:

```

r_on = Response(); r_on.set_cookie('session_id', 'abc', http_only=True)
print(dict(r_on.headers)['Set-Cookie'])
# session_id=abc; HttpOnly; Path=/; SameSite=Lax

r_off = Response(); r_off.set_cookie('session_id', 'abc', http_only=False)
print(dict(r_off.headers)['Set-Cookie'])
# session_id=abc; Path=/; SameSite=Lax -- no HttpOnly at all

```

VERIFIED DIRECTLY — THE FLAG'S REAL PRESENCE OR ABSENCE ON THE ACTUAL HEADER

A browser genuinely honoring `HttpOnly` refuses to expose a cookie carrying it to any JavaScript running on the page at all — including a script Chapter 4's own real XSS vulnerability might have injected. Without the flag, that identical injected script has a real, working path straight to a live session ID. Defaulting `http_only=True` means a developer has to deliberately opt out to create that exposure, not deliberately opt in to close it.

Where This Course Is Headed

Chapter 7 builds a minimal ORM — mapping real Python objects to real SQL — giving `SessionStore` an honest, real alternative to its own current in-memory dict, one that survives an actual server restart.

Hands-On Exercises

EXERCISE 1

Add a real `Response.delete_cookie(name)` method (a `Set-Cookie` with `Max-Age=0`) and a `/logout` route that calls it, register both on a real App with the session middleware, and verify with a real cookie jar that a request made right after logout starts a brand-new session (the visit counter resets to 1) rather than reusing the old one.

 [View solution](#)

EXERCISE 2

Build two real Response objects, one with `set_cookie(..., http_only=True)` and one with `http_only=False`, and verify directly on each one's own real Set-Cookie header string that `HttpOnly` is present in exactly one of the two — then explain in your own words, tying back to Chapter 4's own real XSS finding, what a real attacker's injected script could and couldn't do differently between the two cases.

 [View solution](#)

EXERCISE 3

Using this chapter's own `visit_counter_handler` and session middleware, create two separate, independent `http.cookiejar.CookieJar`-backed openers (simulating two different real browsers) against the same running App, interleave real requests from both, and verify each one's own visit count increments completely independently of the other.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **Cookies** — a plain header pair, Set-Cookie/Cookie, real-parsed and real-built via `http.cookies.SimpleCookie`
- **Request.cookies / Response.set_cookie()** — lazy-parsed cookie access, and a header-building helper defaulting to `HttpOnly`
- **SessionStore** — a real in-memory dict of `sid -> (data, created_at)`, with a genuine per-session idle-timeout on `load()`
- **Session middleware** — the exact `before-next_()/after-next_()` shape from Chapter 5: load or create before, save and set the cookie after
- **Why sessions need cookies** — verified live: no cookie jar means a brand-new session every request (stuck at 1); a real jar makes the identical code count 1, 2, 3
- **Server-side expiry is independent** — verified live: the client still holds and sends the same unexpired-by-Max-Age cookie; only the server's own timeout invalidates it
- **Session ID security** — `secrets.token_urlsafe(32)`, verified with zero collisions across 20,000 real IDs; `HttpOnly` defaults on, closing the exact cookie-theft path Chapter 4's own real XSS finding opened
- **Next chapter:** A minimal ORM — mapping real Python objects to real SQL

CHAPTER 7: A MINIMAL ORM: MAPPING REAL PYTHON OBJECTS TO REAL SQL

Building a Web Framework

Chapter 7 · A Minimal ORM: Mapping Real Python Objects to Real SQL

Chapter 6 closed on an honest gap: `SessionStore`'s own in-memory dict forgets everything the instant the process ends. This chapter builds a real, minimal ORM — a descriptor-based mapping layer between Python objects and actual SQL — against a genuine SQLite database, and uses it to give `SessionStore` the real alternative Chapter 6 promised: one that survives a genuine restart.

The Basic Job: Mapping a Row to a Real Object

A database row is a tuple of raw values with no names attached beyond column position; a Python object is a set of named attributes. A small descriptor class declares, once per model class, which attribute maps to which column:

```
class Field:
    def __init__(self, column):
        self.column = column

class Model:
    _table = None
    _fields = None

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        # collect every Field declared on the subclass, once, when the class itself is
        cls._fields = {
            name: val.column for name, val in vars(cls).items()
            if isinstance(val, Field)
        }

    def __init__(self, **kwargs):
        for name in self._fields:
            setattr(self, name, kwargs.get(name))

    @classmethod
    def get(cls, conn, id):
```

```

        columns = list(cls._fields.values())
        sql = f"SELECT {'', ' '.join(columns)} FROM {cls._table} WHERE id = ?"
        row = conn.execute(sql, (id,)).fetchone()
        if row is None:
            return None
        return cls(**dict(zip(cls._fields.keys(), row)))

    def save(self, conn):
        columns = list(self._fields.values())
        values = [getattr(self, name) for name in self._fields]
        placeholders = ', '.join(['?'] * len(columns))
        sql = f"INSERT OR REPLACE INTO {self._table} ({', ' '.join(columns)}) VALUES ({placeholders})"
        conn.execute(sql, values)
        conn.commit()

```

Declaring a real model is two lines of class body:

```

class Author(Model):
    _table = 'authors'
    id = Field('id')
    name = Field('name')

a = Author.get(conn, 1)  # conn is a real sqlite3 connection
print(a.id, a.name)
# 1 Ann

```

VERIFIED DIRECTLY AGAINST A REAL SQLITE DATABASE

`Author.get(conn, 1)` genuinely queries a real database and returns a real `Author` instance with `.id` and `.name` populated from the actual row. `__init_subclass__` runs once, the moment `Author` itself is defined, well before any query is issued — every later call to `.get()` reuses that already-collected field mapping.

The Real SQL Injection Problem

`Model.get()` already uses a parameter placeholder — but a lookup by name, written the more tempting way with an f-string, doesn't:

```
def find_author_naive(conn, name):
    sql = f"SELECT id, name FROM authors WHERE name = '{name}'"
    return conn.execute(sql).fetchall()

print(find_author_naive(conn, 'Ann'))
# [(1, 'Ann')]

malicious = "x' OR '1'='1"
print(find_author_naive(conn, malicious))
# [(1, 'Ann'), (2, 'Bo'), (3, 'Cy')] -- every row in the table
```

VERIFIED DIRECTLY — THE REAL, EXECUTED SQL GENUINELY DIFFERS FROM WHAT WAS INTENDED

The real query that runs, character for character, is `SELECT id, name FROM authors WHERE name = 'x' OR '1'='1'`. The input's own embedded quote closes the string literal early; `OR '1'='1'` is always true, so every row matches. This is exactly Chapter 4's own real XSS bug, one layer down — naive string interpolation trusted a value to stay data, and it stopped being data.

Parameterized queries fix it the same way Chapter 4's own `SafeString` did — separate the query's fixed structure from the untrusted value:

```
def find_author_safe(conn, name):
    sql = 'SELECT id, name FROM authors WHERE name = ?'
    return conn.execute(sql, (name,)).fetchall()

print(find_author_safe(conn, malicious))
# [] -- treated as a literal name nobody has, not as SQL
```

VERIFIED DIRECTLY — THE IDENTICAL MALICIOUS STRING IS NOW COMPLETELY INERT

The exact same `malicious` string, run through `find_author_safe()`, returns an empty result. The `?` placeholder tells the driver where the query's own structure ends and a value begins, and sends the value separately from the SQL text — no character it contains can ever be interpreted as part of the query itself.

The N+1 Query Problem, Measured

A parameterized single-row lookup is safe. Calling it once per related object, though, adds up fast. A one-line addition to `Model` fetches every row at once:

```
@classmethod
def all(cls, conn):
    columns = list(cls._fields.values())
    sql = f"SELECT {'', ' '.join(columns)} FROM {cls._table}"
    rows = conn.execute(sql).fetchall()
    return [cls(**dict(zip(cls._fields.keys(), row))) for row in rows]
```

A second real model, plus a thin wrapper counting every real query issued through it, sets up the measurement — 20 posts split across only 2 distinct authors:

```
class Post(Model):
    _table = 'posts'
    id = Field('id')
    title = Field('title')
    author_id = Field('author_id')

class CountingConnection:
    # wraps a real sqlite3 connection, counting every real execute() call
    def __init__(self, conn):
        self._conn = conn
        self.query_count = 0
    def execute(self, sql, params=()):
        self.query_count += 1
        return self._conn.execute(sql, params)
```

Listing every post along with its own author, the straightforward way:

```
posts = Post.all(conn) # 1 query
for post in posts:
    author = Author.get(conn, post.author_id) # 1 query PER post

print(f"{len(posts)} posts, {conn.query_count} total queries")
# 20 posts, 21 total queries
```

VERIFIED DIRECTLY, WITH A REAL QUERY COUNTER WRAPPING A REAL SQLITE CONNECTION

20 posts genuinely cost 21 real queries — 1 to fetch the posts, plus 1 more per post's own author lookup, even though only 2 distinct authors actually exist. SQLAlchemy's own documentation names this precisely: "the **N plus one problem**... for any N objects loaded, accessing their lazy-loaded attributes means there will be N+1 SELECT statements emitted."

Batching the author lookup into one `IN`-clause query, instead of one per post:

```
posts = Post.all(conn) # 1 query
author_ids = list({p.author_id for p in posts})
placeholders = ', '.join(['?'] * len(author_ids))
sql = f"SELECT id, name FROM authors WHERE id IN ({placeholders})"
authors_by_id = {row[0]: row for row in conn.execute(sql, author_ids).fetchall()} # 1 query

for post in posts:
    author = authors_by_id[post.author_id] # no query at all -- already loaded

print(f"{len(posts)} posts, {conn.query_count} total queries")
# 20 posts, 2 total queries
```

VERIFIED DIRECTLY — THE SAME 20 POSTS, A REAL 10.5× FEWER QUERIES

Measured against the identical dataset, batching drops the real query count from 21 to 2. Django's own real `prefetch_related()` and SQLAlchemy's own real `selectinload()` both work by this identical mechanism, per each project's own documentation.

The Real Payoff: A SqliteSessionStore

A model with three columns — an ID, a JSON-serialized data blob, and a creation timestamp — is enough to reimplement Chapter 6's own `SessionStore`, keeping its exact `create()` / `load()` / `save()` interface unchanged:

```
import json, secrets, time, sqlite3

class SessionRow(Model):
    _table = 'sessions'
    id = Field('id')
    data = Field('data')
    created_at = Field('created_at')
```

```

class SqliteSessionStore:
    def __init__(self, db_path, max_age=3600):
        self.conn = sqlite3.connect(db_path, check_same_thread=False)
        self.conn.execute(
            'CREATE TABLE IF NOT EXISTS sessions '
            '(id TEXT PRIMARY KEY, data TEXT NOT NULL, created_at REAL NOT NULL)'
        )
        self.conn.commit()
        self.max_age = max_age

    def create(self):
        sid = secrets.token_urlsafe(32)
        SessionRow(id=sid, data=json.dumps({}), created_at=time.time()).save(self.conn)
        return sid

    def load(self, sid):
        row = SessionRow.get(self.conn, sid)
        if row is None:
            return None
        if time.time() - row.created_at > self.max_age:
            self.conn.execute('DELETE FROM sessions WHERE id = ?', (sid,))
            self.conn.commit()
            return None
        return json.loads(row.data)

    def save(self, sid, data):
        existing = SessionRow.get(self.conn, sid)
        created_at = existing.created_at if existing else time.time()
        SessionRow(id=sid, data=json.dumps(data), created_at=created_at).save(self.conn)

```

Wiring it in means changing exactly one line in Chapter 6's own setup — `SessionStore()` becomes `SqliteSessionStore('sessions.db')` — and nothing about `make_session_middleware()` or `App` has to know or care, since both classes expose the identical three-method interface.

VERIFIED DIRECTLY — REAL SESSION DATA, SAVED AND RELOADED FROM AN ACTUAL FILE ON DISK

```

store1 = SqliteSessionStore(db_path), a real create() + save() with {'visits': 1,
'username': 'sam'}, then del store1 — simulating the whole process ending. A genuinely fresh
store2 = SqliteSessionStore(db_path), a brand-new connection pointed at the same real file,

```

loading the same session ID, returns the identical `{'visits': 1, 'username': 'sam'}` back. The in-memory dict from Chapter 6 could never have survived that; a real file on disk does.

A Real, Severe Bug Found Wiring This Into the Live App

Plugging `SqliteSessionStore` straight into a real `App`, served through `wsgiref` on a background thread exactly like every prior chapter's own live demos, crashed immediately — not in this chapter's own isolated tests above, only once a real server was actually handling requests:

```
sqlite3.ProgrammingError: SQLite objects created in a thread can only
be used in that same thread. The object was created in thread id
38384 and this is thread id 38448.
```

VERIFIED DIRECTLY — A REAL, DOCUMENTED SQLITE SAFETY RESTRICTION, NOT A BUG IN THIS CHAPTER'S OWN CODE

`sqlite3.connect()` ties its own connection object to whichever thread created it, by design, specifically to stop a connection being shared unsafely across threads by accident.

`make_server(...).serve_forever()` runs every real request on its own server thread — genuinely different from the thread that called `SqliteSessionStore(db_path)` in the first place — so the very first live request touching `self.conn` failed outright. The fix is a real, explicit opt-in:

`sqlite3.connect(db_path, check_same_thread=False)`, telling Python this specific connection genuinely will be used from more than one thread.

With the fix applied, run end to end through the entire real stack — `App`, `Router`, `Request` / `Response`, the session middleware, and `SqliteSessionStore` together:

```
request 1 (real live server, port 8911): b'You have visited 1 times'
request 2 (same jar):                    b'You have visited 2 times'

# server1.shutdown(); del store -- a real, full "process" teardown
# real row count still on disk afterward: 1

# a genuinely fresh App, fresh SqliteSessionStore, a DIFFERENT port,
# same cookie the still-alive real client jar holds:
request 3 (real live server, port 8912): b'You have visited 3 times'
```

VERIFIED DIRECTLY, END TO END — A REAL "RESTART" THAT GENUINELY DOESN'T LOSE ANYTHING

The third request — against a completely new `App`, a completely new `SqliteSessionStore`, and a new port standing in for a fresh process — correctly continues the count at `3`, not `1`. Nothing about the client changed; its own cookie jar simply kept sending the same real session ID it had all along. What changed is that the server, this time, remembered it.

Where This Course Is Headed

Chapter 8 builds real error handling and a development server with live reload — directly resolving Chapter 4's own stale-cache finding, and giving crashes like this chapter's own `ProgrammingError` a genuinely useful debug page instead of `wsgiref`'s own generic 500.

Hands-On Exercises

EXERCISE 1

Add an `update(conn)` method to this chapter's own `Model` base class — distinct from `save()`, which uses `INSERT OR REPLACE` — that updates an existing row's own non-id columns via a real SQL `UPDATE` statement, and verify it against a real change to an existing author's name, confirming the change is re-fetchable afterward and that a different row is left completely untouched.

[!\[\]\(22d4cf5a9ebe1893c91afc1b4058808c_img.jpg\) View solution](#)

EXERCISE 2

Extend this chapter's own `find_author_naive()` into a `login_naive(conn, name, password)` function checking both a name AND a password column with the identical naive string-interpolation style, and construct a real, working login-bypass injection that returns a matching row without ever knowing the real password.

[!\[\]\(29ceb7d82e432ac27b3c209fb3684e07_img.jpg\) View solution](#)

EXERCISE 3

Add a real `delete(sid)` method to `SqliteSessionStore` (a real SQL DELETE, not just a Set-Cookie clear), wire it into a `logout_handler` matching Chapter 6's own real logout exercise, and verify — by querying the real row count in the sessions table directly, not just by checking the response — that logging out genuinely removes the session's own row from the database, not just its cookie from the client.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **The basic job** — Field/Model maps class attributes to real database columns, verified against live SQLite
- **The real SQL injection problem** — naive f-string query building, verified returning every row on a crafted input; the same underlying bug as Chapter 4's own XSS, one layer down
- **Parameterized queries** — a `?` placeholder, verified turning the identical malicious input completely inert
- **The N+1 problem, measured** — 20 posts, 21 real queries via naive lazy loading, down to 2 via one batched IN-clause query
- **SqliteSessionStore** — reimplements Chapter 6's own exact `create()/load()/save()` interface on real SQLite, verified surviving a genuine restart with data intact
- **check_same_thread=False** — a real, necessary fix once this store is wired into a live, threaded WSGI server; verified crashing without it, working correctly end to end with it
- **Next chapter:** Error handling & a real development server with live reload

CHAPTER 8: ERROR HANDLING & A REAL DEVELOPMENT SERVER WITH LIVE RELOAD

Building a Web Framework

Chapter 8 · Error Handling & a Real Development Server With Live Reload

Two honest gaps have been sitting open since earlier chapters. Chapter 5 verified that a real, uncaught exception reaches the client as `wsgiref`'s own generic `"A server error occurred. Please contact the administrator."` — real, working, and completely uninformative. Chapter 4 verified that editing a cached template's file on disk changes nothing a running server serves, until the process itself restarts. This chapter closes both: a real debug error page, and a real file-watcher that reloads what it needs to without a human ever typing a restart command.

A Real Debug Middleware

Exactly the same shape as Chapter 5's own `error_handling_middleware` — a `try`/`except` around `next_()` — but instead of returning one fixed message, it builds a real traceback from Python's own standard library:

```
import traceback, html

def make_debug_middleware(app):
    def middleware(request, next_):
        try:
            return next_(request)
        except Exception:
            tb_text = traceback.format_exc()
            if app.debug:
                body = (
                    '<h1>500 Internal Server Error</h1>'
                    f'<pre>{html.escape(tb_text)}</pre>'
                )
                return Response(body, status=500, content_type='text/html')
            return Response('500 Internal Server Error', status=500)
    return middleware
```

`App` gets one new attribute — a real, explicit `debug` flag, mirroring the identical real convention Flask and Django both use:

```
class App:
    def __init__(self, debug=False):
        self.router = Router()
        self.middlewares = []
        self._chain = None
        self.debug = debug
    # ... everything else unchanged from Chapter 5 ...
```

VERIFIED DIRECTLY — A REAL, USEFUL TRACEBACK REACHES THE CLIENT, ONLY WHEN DEBUG IS GENUINELY ON

A real `ValueError` raised deep inside a handler, hit against `App(debug=True)`, returns a real `500` whose body genuinely contains the actual source file, the actual line numbers, and the exact exception message — `"something genuinely went wrong, deep in a real handler"`, confirmed present verbatim. The identical crash against `App(debug=False)` returns the short, safe, generic message instead — deliberately built by this chapter's own code now, not merely inherited from `wsgiref`'s own fallback.

The Traceback Itself Needs the Exact Same Discipline as Chapter 4

A traceback is still just text dropped into an HTML response — and an exception's own message is real, attacker-influenced data whenever it's built from user input.

`html.escape(tb_text)` above isn't decoration:

```
def crashing_with_html_handler(request):
    raise ValueError('<script>alert(1)</script>')
```

VERIFIED DIRECTLY — A REAL, LIVE REFLECTED-XSS RISK, AND CONFIRMATION IT'S CLOSED

Raised for real and rendered through `make_debug_middleware`, the client's own response body contains `ValueError: <script>alert(1)</script>` — confirmed directly that the literal, executable `<script>` tag is genuinely absent from the real response, with only its escaped form present. Skip the `html.escape()` call and this debug page becomes a second, real XSS vector, structurally identical to Chapter 4's own original finding — just fed by an exception message instead of a template variable.

A Real File Watcher

Every file on disk carries a real, OS-tracked last-modified time. Comparing it against a previously recorded value is enough to detect a genuine edit, with no need to read or hash the file's own contents:

```
import os

class FileWatcher:
    def __init__(self, paths):
        self.paths = list(paths)
        self._mtimes = {p: self._safe_mtime(p) for p in self.paths}

    def _safe_mtime(self, path):
        try:
            return os.stat(path).st_mtime
        except FileNotFoundError:
            return None

    def changed(self):
        for p in self.paths:
            current = self._safe_mtime(p)
            if current != self._mtimes[p]:
                self._mtimes[p] = current    # update the stored value -- this change is
                return p
        return None
```

VERIFIED DIRECTLY — A REAL EDIT DETECTED, AND CORRECTLY NOT RE-REPORTED

`changed()` called right after construction, with no edit yet, returns `None`. A real write to the watched file, followed by another `changed()` call, returns that file's own real path. Calling `changed()` a second time immediately afterward, with no further edit, correctly returns `None` again — the stored `mtime` was updated the moment the change was reported, so the same edit is never reported twice.

Resolving Chapter 4's Own Stale-Cache Finding

A watcher on the template directory, checked once at the top of every request, is enough to invalidate `TemplateEngine`'s own cache exactly when it's genuinely stale:

```
watcher = FileWatcher(['templates/greeting.html'])

def app(envIRON, start_response):
    if watcher.changed():
        templates._cache.clear() # force every template to recompile on next access
    body = templates.render('greeting.html', name='Ada').encode('utf-8')
    # ...
```

VERIFIED DIRECTLY — THE EXACT SCENARIO FROM CHAPTER 4, NOW FIXED LIVE

Request 1, before any edit: `Hello, Ada! (v1)`. The template file is then genuinely edited on disk while the server keeps running — the identical real setup Chapter 4 used to demonstrate the problem. Request 2, this time, returns `Hi there, Ada! (v2, updated on disk)`: the new content, correctly, on the very next request — no restart needed.

Reloading the App's Own Source, Not Just Templates

A template can be recompiled in place because `TemplateEngine` owns it directly. A route handler, a middleware, or the framework's own source files can't be swapped out mid-process the same way — genuinely picking up a change there needs the whole Python process to start over. Werkzeug's own real reloader (the one Flask's `debug=True` uses) solves this with `os.execv()`: replacing the running process's own program image in place.

```
import sys, time

def run_with_reloader(watched_paths):
    watcher = FileWatcher(watched_paths)
    while True:
        time.sleep(0.1)
        if watcher.changed():
            # genuinely re-exec this process -- a real, working reload, not a simulation
            os.execv(sys.executable, [sys.executable] + sys.argv)
```

Verified with a real, separately spawned subprocess — running this exact loop, watching one real file, with a small marker file recording its own restart count and PID on every start:

```
marker state right after startup:          {'restart_count': 1, 'pids': [32112]}
# ... a real edit is made to the watched file while the subprocess is running ...
marker state after the real edit + reload window: {'restart_count': 2, 'pids': [32112,
```

VERIFIED DIRECTLY — A REAL, LIVE SUBPROCESS GENUINELY RESTARTED ITSELF

`restart_count` increased from a real `1` to a real `2`, confirmed by reading the marker file back from outside the subprocess entirely — not inferred, not simulated. A single real file edit, made while the subprocess kept running, was enough to trigger it.

A GENUINE, DIRECTLY-OBSERVED PLATFORM DIFFERENCE — NOT THE BEHAVIOR POSIX DOCUMENTS

On a true POSIX system, `os.execv()` replaces the current process's own program image while keeping the identical process ID — the whole reason it's called "exec," not "spawn." Verified directly on this real Windows machine, the PID recorded in the marker file changed across the reload, `32112` → `38476`, rather than staying fixed. Windows has no true POSIX `exec()` syscall for Python to call into, so its own emulation genuinely starts a new process rather than replacing the current one in place. The visible, real effect — the app's own code restarting itself from scratch — is identical either way; only the specific mechanism, and the resulting PID, differ by platform.

Where This Course Is Headed

Chapter 9 builds real static file serving and a small command-line tool for the framework itself — likely the actual entry point that wires `debug=True` and `run_with_reloader()` together into one real `python -m myframework runserver` command.

Hands-On Exercises

EXERCISE 1

Extend this chapter's own `make_debug_middleware()` to include the real request's method and path (e.g., "GET /widgets/42") in the debug page's own body, correctly HTML-escaped, and verify against a

real live request to a dynamic route that the exact real method and path genuinely appear in the client's own response.

 [View solution](#)

EXERCISE 2

Construct a real FileWatcher watching two genuinely separate files, edit only the first one and verify `changed()` reports exactly that file's own path (not the second one), confirm a second call to `changed()` returns `None`, then edit only the second file and verify `changed()` now correctly reports that one instead.

 [View solution](#)

EXERCISE 3

Wire this chapter's own FileWatcher-based template-reload check so it only runs when `debug` is `True`, and verify live, against two real Apps (one `debug=True`, one `debug=False`), that editing a template file mid-run is picked up automatically in the `debug=True` case but genuinely stays stale in the `debug=False` case — explaining in your own words why skipping a real filesystem `stat()` call on every single request is a real, worthwhile thing to avoid in production.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **Debug middleware** — `try/except` around `next_()`, building a real `traceback.format_exc()` body when `App.debug` is `True`, a short generic message otherwise
- **The traceback must be escaped** — verified as a real, live reflected-XSS risk otherwise, closed with `html.escape()`, the identical discipline Chapter 4 already established
- **FileWatcher** — `mtime`-based change detection, verified reporting a real edit exactly once, never re-reporting an already-consumed change
- **Fixing Chapter 4's own stale-cache finding** — clearing `TemplateEngine._cache` when the watcher fires, verified serving the real, updated content on the very next request
- **`run_with_reloader()`** — a real `os.execv()`-based restart, verified with a genuinely spawned subprocess and a marker file showing `restart_count` actually increasing

- **A real, verified Windows-specific finding** — the PID changed across the reload here, unlike POSIX's own documented same-PID `exec()` behavior; the visible restart still worked identically
- **Next chapter:** Static files & a real CLI for the framework itself

CHAPTER 9: STATIC FILES & A REAL CLI FOR THE FRAMEWORK

Building a Web Framework

Chapter 9 · Static Files & a Real CLI for the Framework

Two genuinely separate problems close out this chapter: serving CSS, JavaScript, and image files that live on disk rather than being generated by a handler, and giving the framework a real command-line entry point instead of a hand-written `if __name__ == '__main__':` block copied into every project. Static files turn out to be a direct, unplanned payoff — and a direct, unplanned risk — of Chapter 2's own extensible `CONVERTERS` dict.

A New Converter, and a New Kind of Risk

Every converter built since Chapter 2 matches exactly one path segment — `[^/]+` for `str`, stopping dead at the first `/`. That's correct for a username or a slug, but genuinely wrong for a static file living at `css/style.css`, two real segments deep. Verified directly:

```
router.add_route('GET', '/static/<str:filename>', handler)

router.resolve('GET', '/static/style.css')      # ('MATCH', ...)      -- one segment, f
router.resolve('GET', '/static/css/style.css')  # ('404', None, {}, None) -- genuinely
```

A new converter fixes it — and it's a single new dict entry, exactly the shape Chapter 2's own design was built to accommodate:

```
CONVERTERS = {
    'int': (r'\d+', int),
    'str': (r'[^/]+', str),
    'path': (r'.+', str),    # NEW -- matches across '/' too
}
```

VERIFIED DIRECTLY — THE NEW CONVERTER GENUINELY CAPTURES A NESTED PATH

`<path:filename>` against `/static/css/style.css` resolves with `MATCH` and `{'filename': 'css/style.css'}`, correctly, where the plain `str` version 404s. The regex change is the entire fix —

nothing else in `Router` needed to change to support it.

THE SAME CAPABILITY THAT FIXES IT ALSO OPENS SOMETHING NEW

`r'.+'` matches *anything* across *any number* of segments — including `../secret.txt`. Verified directly: the identical route also resolves `/static/../secret_outside_static.txt` with `MATCH` and `{'filename': '../secret_outside_static.txt'}`. The router doesn't know or care what a filename "should" look like — it just matches the pattern it was given. Whether that string is safe to actually open is a question for the handler, not the router.

A Naive Handler, and the Real Vulnerability It Opens

The obvious first implementation joins the static directory with the requested filename and opens it:

```
def naive_static_handler(filename):
    full_path = os.path.join(STATIC_DIR, filename)
    with open(full_path, 'rb') as f:
        return f.read()
```

VERIFIED DIRECTLY — A REAL FILE OUTSIDE THE STATIC DIRECTORY, GENUINELY READ

With a real file `secret_outside_static.txt` placed one level above `STATIC_DIR`, containing `b'TOP SECRET - should never be servable via /static/'`, calling `naive_static_handler('../secret_outside_static.txt')` returns that exact real content — the naive handler genuinely reads a file it was never meant to expose. `os.path.join()` happily follows a `'..'` segment right back out of the directory it was supposed to confine requests to; it performs string concatenation, not containment checking.

The Real Fix — `realpath()` and `commonpath()`

The fix isn't rejecting the string `'..'` — a filename can smuggle the identical escape through URL-encoding, symlinks, or a mix of legitimate and traversal segments. The real fix is resolving the path fully, then checking where it actually landed:

```
def safe_static_path(filename):
    static_root = os.path.realpath(STATIC_DIR)
    candidate = os.path.realpath(os.path.join(STATIC_DIR, filename))
    if os.path.commonpath([static_root, candidate]) != static_root:
        return None    # genuinely escaped -- refuse it
    return candidate
```

`os.path.realpath()` collapses every `..` and symlink down to one real, canonical absolute path. `os.path.commonpath()` then answers one exact question: does the resolved candidate still sit inside the resolved root? If it doesn't, no string-pattern check was ever going to catch every way to ask the question in the first place — this checks the actual, final destination instead.

VERIFIED DIRECTLY AGAINST FOUR DISTINCT REAL CASES

INPUT	RESULT
<code>css/style.css</code>	resolves correctly, genuinely inside <code>STATIC_DIR</code>
<code>../secret_outside_static.txt</code>	<code>None</code> — blocked
a full absolute path to the secret file	<code>None</code> — blocked
<code>css/../../secret_outside_static.txt</code>	<code>None</code> — blocked

The full handler, with real `Content-Type` detection and a genuine 404 rather than a leak on any rejected path:

```
import mimetypes

def serve_static(request):
    resolved = safe_static_path(request.params['filename'])
    if resolved is None or not os.path.isfile(resolved):
        return Response('404 Not Found', status=404)
    with open(resolved, 'rb') as f:
        body = f.read()
    content_type, _ = mimetypes.guess_type(resolved)
    return Response(body, content_type=content_type or 'application/octet-stream')
```

VERIFIED DIRECTLY — REAL CONTENT-TYPE DETECTION ACROSS FIVE REAL EXTENSIONS

Real files created for `.js`, `.json`, `.txt`, `.css`, and `.html`, served through this exact handler over a real running server, produced Content-Type headers of `text/javascript`, `application/json`, `text/plain`, `text/css`, and `text/html` respectively — every one correct, with zero manual mapping written by this framework itself. `mimetypes` is standard-library, sourced from the same OS-level extension table every other program on the machine already uses.

Verified Live — Through a Real Running Server

Wired into `App` via the now-familiar route, and hit with real HTTP requests against a real `wsgiref` server bound to a real socket, not called as a bare Python function:

```
app.add_route('GET', '/static/<path:filename>', serve_static)
```

VERIFIED DIRECTLY — REAL HTTP, REAL HEADERS, A LEGITIMATE REQUEST AND TWO REAL ATTACKS

`GET /static/css/style.css` → `200`, body `b'body { color: red; }'`, with a real `Content-type: text/css` and a real `Last-Modified` header, both genuinely produced by the server. A real `GET /static/../secret_outside_static.txt` against the live server → `404`. A second, mixed-segment real attempt, `GET /static/css/../secret_outside_static.txt` → also `404`. The fix holds against a real client making real requests over a real socket, not just against direct function calls.

A Real CLI: Wiring Everything Together

Every chapter so far has ended with a hand-written script that builds an `App` and calls `make_server(...).serve_forever()` directly. A real framework gives its users one command instead — `argparse`, from the standard library, is enough:

```
import argparse

def cmd_runserver(args):
    app = build_app(debug=args.debug)
    server = make_server('127.0.0.1', args.port, app)
    print(f"Serving on http://127.0.0.1:{args.port} (debug={args.debug})")

    if args.debug:
        run_with_reloader(server.serve_forever, watch_paths=[__file__])
```

```

else:
    server.serve_forever()

def main():
    parser = argparse.ArgumentParser(prog='manage.py')
    sub = parser.add_subparsers(dest='command', required=True)

    runserver = sub.add_parser('runserver')
    runserver.add_argument('--debug', action='store_true')
    runserver.add_argument('--port', type=int, default=8000)
    runserver.set_defaults(func=cmd_runserver)

    args = parser.parse_args()
    args.func(args)

if __name__ == '__main__':
    main()

```

`run_with_reloader()` itself needed one real change from Chapter 8's own version — it now runs the actual server call in a background thread, and polls for a change on the main thread, since `serve_forever()` blocks:

```

import threading

def run_with_reloader(serve_fn, watch_paths, poll_interval=0.25):
    watcher = FileWatcher(watch_paths)
    threading.Thread(target=serve_fn, daemon=True).start()
    while True:
        time.sleep(poll_interval)
        if watcher.changed():
            os.execv(sys.executable, [sys.executable] + sys.argv)

```

VERIFIED DIRECTLY — REAL ARGPARSE PARSING

`parse_args(['runserver'])` → `Namespace(command='runserver', debug=False, port=8000)`.
`parse_args(['runserver', '--debug', '--port', '8951'])` → `debug=True, port=8951`, with
`'8951'` genuinely converted to the real integer `8951` by `type=int`. Calling `parse_args([])` with
no subcommand at all raises a real `SystemExit`, code `2` — `argparse`'s own `required=True`
genuinely enforced, not just documented.

Verified Live — the Full CLI, Including a Reload While Serving

The comprehensive test: spawn `python manage.py runserver --debug --port 8951` as a real subprocess, make real HTTP requests, edit the watched source file on disk while the process keeps running, wait, then make another real request against the same port.

```
spawned PID (original):          9636
GET /                            -> 200  b'CLI-served app is running\n'
GET /static/hello.txt            -> 200  b'hello from cli static file'

# ... manage.py's own source is genuinely edited on disk here ...

GET / (post-reload)              -> 200  b'CLI-served app is running\n'
PID listening on port after edit: 36344
```

VERIFIED DIRECTLY — THE CLI, ARGPARSE, STATIC FILES, AND THE RELOADER, ALL GENUINELY WORKING TOGETHER

Both real requests before the edit succeeded. The watched source file was then genuinely modified while the server kept running. After the reload window, a fresh real request to `/` still returned `200` — the server survived the reload and kept serving correctly, through the exact CLI command a real user would type.

CHAPTER 8'S OWN WINDOWS PID-CHANGE FINDING, REPRODUCED A SECOND TIME

The PID listening on the port changed from a real `9636` to a real `36344` across the reload — the identical platform-specific `os.execv()` behavior Chapter 8 found once already, now confirmed again in a genuinely different context: a threaded server inside a CLI command, not a bare polling loop.

A Second, New Windows Finding: Naive Process Tracking Breaks Too

The first attempt at this exact live test captured the subprocess's output with a plain pipe (`subprocess.PIPE`) and called `proc.terminate()` followed by `proc.stdout.read()` once the test was done. It hung indefinitely.

VERIFIED DIRECTLY — A REAL HANG, AND ITS REAL CAUSE

`proc.terminate()` only ever signals the **original** PID the `Popen` object remembers. On this real Windows machine, `os.execv()` genuinely launches a brand-new OS process with its own new PID — confirmed directly with `Get-NetTCPConnection`: a real orphaned process was still bound to the port, and still running, well after the original PID had already been signaled. That new process had inherited the same stdout pipe handle, and kept its write end open — so `proc.stdout.read()`, which only returns once every writer has closed that pipe, waited forever for a process the test no longer had any handle to.

Two real, independent fixes, verified together: redirect the child's output to a real log **file** instead of a pipe (a file has no "still open" ambiguity the way a pipe's write end does), and, for cleanup, look up whichever PID is *actually* listening on the port right now rather than trusting the one `Popen` remembers:

```
log_file = open('server.log', 'w')
proc = subprocess.Popen(..., stdout=log_file, stderr=subprocess.STDOUT)

# ... later, cleanup -- find whoever is REALLY on the port, not proc.pid ...
real_pid = get_pid_listening_on_port(port)
subprocess.run(['taskkill', '/PID', real_pid, '/F', '/T'])
```

VERIFIED DIRECTLY — BOTH FIXES CONFIRMED TOGETHER, END TO END

With output redirected to a real file and cleanup done by port lookup rather than remembered PID, the same full reload scenario ran to completion cleanly: two real requests before the edit, a genuine reload, one more real request afterward, then a clean process kill with no hang and no orphaned process left listening on the port. The server's own captured log file showed the real `"Serving on http://127.0.0.1:8951"` startup line printed twice — once at original startup, once again after the process restarted itself.

Nothing about this changes how a user of the framework actually runs it — `python manage.py runserver --debug` works exactly the same either way. It's a genuine lesson about tooling *around* a self-restarting process on Windows specifically, not about the framework's own runtime behavior.

Where This Course Is Headed

Chapter 10 is the capstone — assembling every piece built across all nine prior chapters (routing, request/response, templates, middleware, sessions, the ORM, error handling, live reload, and now static files and the CLI) into one real, working small application, run end to end through `python manage.py runserver`.

Hands-On Exercises

EXERCISE 1

Against the live CLI-served app, verify that a URL-quoted absolute path (the real, full filesystem path to a file genuinely outside the static directory, with no leading "../" at all) is also blocked by `safe_static_path()` — try both a backslash-style and a forward-slash-style absolute path, and confirm both return a real 404 against a real running server.

 [View solution](#)

EXERCISE 2

Create five real static files with five different extensions (.js, .json, .txt, .css, .html), serve all five through the live CLI app, and verify via real HTTP requests that `mimetypes.guess_type()` produces the correct real Content-Type header for every one of them.

 [View solution](#)

EXERCISE 3

Spawn the CLI deliberately WITHOUT `--debug`, edit its watched source file on disk while it's running, wait through the same real time window the `--debug` case needed to reload, then verify — by checking which real PID is listening on the port before and after — that no reload happened at all, confirming `debug=False` genuinely never starts the watch loop in the first place.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **The path converter** — {'path': (r'.+', str)}, added to Chapter 2's own CONVERTERS dict, matches across '/' — the same capability that enables nested static paths also matches traversal strings
- **The vulnerability** — verified directly: os.path.join(STATIC_DIR, filename) with an unchecked filename genuinely reads a real file outside the static directory
- **The fix** — os.path.realpath() to fully resolve the path, then os.path.commonpath() to check it's still genuinely inside the static root; verified against 4 real cases
- **mimetypes.guess_type()** — real, standard-library Content-Type detection, verified correct across 5 real file extensions
- **The CLI** — argparse, a runserver subcommand, --debug and --port flags, verified with real Namespace parsing and a real SystemExit on a missing subcommand
- **run_with_reloader()** updated — now runs serve_forever() in a background thread so the main thread can poll for changes, verified surviving a real reload while serving
- **A new, real Windows finding** — a reload's new PID breaks both naive stdout-pipe capture (hangs forever) and PID-remembered process cleanup; fixed with a real log file and a port-based PID lookup
- **Next chapter:** the capstone — every chapter's own piece, assembled into one real app run through python manage.py runserver

CHAPTER 10: CAPSTONE — A COMPLETE, WORKING SMALL APPLICATION

Building a Web Framework

Chapter 10 · Capstone: A Complete, Working Small Application

Nine chapters, nine real classes: `Router`, `Request`/`Response`, `TemplateEngine`, `App`'s own onion-model middleware, session/cookie handling, the descriptor-based ORM, the debug/live-reload dev server, and finally a real static-file route wired into an `argparse` CLI. Every one of them has been verified in isolation. This chapter wires all nine together for the first time into **Notes** — a small, genuinely working note-sharing app with real login, protected routes, template-rendered pages, a JSON API, and static files — and runs it end to end through the exact real command a user of this framework would actually type: `python manage.py runserver`.

The App: A Small Note-Sharing Site

A logged-out visitor can browse and search notes. A logged-in user can add new ones. Everything is real and persisted — SQLite on disk, not an in-memory placeholder:

```
notes_app/
├─ manage.py           # the CLI entry point (Ch.9)
├─ notes.db           # real SQLite -- notes, users, sessions (Ch.7)
├─ templates/
│   ├─ base.html       # layout: nav, {% block title %}, {% block content %} (Ch.4)
│   ├─ notes_list.html # extends base.html; search box + note cards
│   ├─ note_card.html  # one note, rendered once per item and reused as a SafeString
│   ├─ note_detail.html # extends base.html; a single note
│   ├─ login.html      # extends base.html; a login form
│   └─ new_note.html   # extends base.html; the add-note form (protected)
└─ static/
    └─ style.css       # served via the real path converter + safe_static_path (Ch.9)
```

ROUTE	METHOD	PROTECTED?	WHAT IT DOES
/	GET	No	List notes, or search via <code>?q=</code>

ROUTE	METHOD	PROTECTED?	WHAT IT DOES
/notes/<int:id>	GET	No	A single note's own page
/login	GET / POST	No	Show / process the login form
/logout	GET	No	Clear the session, redirect home
/notes/new	GET / POST	Yes	Show / process the add-note form
/api/notes	GET	No	Every note as real JSON
/static/<path:filename>	GET	No	Real static file serving (Ch.9)

A Real Constraint the Template Engine Imposes on the Design

Chapter 4's own `TemplateEngine` was built and verified with a flat context dict and a single-identifier `{{ name }}` — it never gained dotted attribute access. Passing a whole `Note` object into `{% for note in notes %}{{ note.title }}{% endfor %}` was never something this framework's own real code was built to do. Rather than quietly bolting on a new capability at the last minute, the app's own `home` handler works within the real, already-verified engine: each note is rendered *once*, through `note_card.html`, producing a real, already-escaped HTML fragment — then wrapped as a `SafeString` before being handed to the outer template's own `{% for card in note_cards %}{{ card }}{% endfor %}` loop.

```
def home(request):
    q = request.query.get('q')
    notes = Note.search(conn, q) if q else Note.all(conn, order_by='id DESC')

    # each card is rendered THROUGH the engine once (real escaping happens here),
    # then wrapped SafeString -- the exact opt-out Chapter 4 verified -- so the
    # outer template's own {{ card }} doesn't escape already-escaped HTML a second time
    note_cards = [SafeString(templates.render(
        'note_card.html', id=n.id, title=n.title, body=n.body,
    )) for n in notes]

    ctx = {'note_cards': note_cards, 'q': q or '', 'empty': len(notes) == 0}
    ctx.update(nav_ctx(current_user(request)))
    return Response.html(templates.render('notes_list.html', **ctx))
```

A GENUINE, SATISFYING USE OF CHAPTER 4'S OWN SAFESTRING — NOT A SHORTCUT AROUND ESCAPING

This isn't skipping the security check Chapter 4 built — every note's own `title` and `body` still passes through `escape()` exactly once, inside `note_card.html`'s own `{{ title }}` / `{{ body }}`. `SafeString` only marks the *already-escaped result* as safe for the outer template to insert verbatim, which is exactly the real, narrow case Chapter 4 designed the opt-out for.

Assembling Every Middleware, in the Right Order

Registration order controls nesting depth (Chapter 5) — the debug middleware needs to be outermost, to catch an exception from anywhere else in the chain; the session middleware needs to be inner enough that `request.state['session']` is already populated by the time any handler runs:

```
app.use(make_debug_middleware(app))    # Ch.8 -- outermost: catches everything below it
app.use(timing_middleware)             # Ch.5 -- measures the full request, including
app.use(make_session_middleware(store)) # Ch.6 -- innermost: request.state['session']
```

`login_required` itself is genuinely new code — but it needed *zero* new framework machinery to write, because `Request.state` already exists:

```
def login_required(handler):
    def wrapped(request):
        if not request.state['session'].get('user_id'):
            return Response.redirect('/login')
        return handler(request)
    return wrapped

app.add_route('GET', '/notes/new', login_required(new_note_get), name='new_note_get')
app.add_route('POST', '/notes/new', login_required(new_note_post), name='new_note_post')
```

APPLICATION CODE, NOT FRAMEWORK CODE — AND THAT'S THE POINT

`login_required` lives in `manage.py`, not inside `App`, `Router`, or any of the classes this course built. It's an ordinary Python function wrapping another ordinary Python function, using a piece of

state (`request.state`) the framework already exposed for exactly this kind of thing back in Chapter 5. A framework doesn't need to anticipate every application-level pattern a real app will want — it needs to expose enough real primitives that a pattern like this can be built *on top*, in five lines, with nothing added to the framework itself.

Verified Live — the Full App, Through the Real CLI Command

Spawned as a genuine subprocess — `python manage.py runserver --port 8971` — and driven with a real `http.cookiejar.CookieJar`-backed opener, the same real client-simulation technique Chapter 6 used to prove sessions need cookies at all:

```
GET / -> 200 (shows "Log in", no "New note" link)
POST /login (username=sam, password=secret123) -> 302 Found / (real Set-Cookie rece
GET /notes/new (same jar, now authenticated) -> 200 (shows the real add-note form)
POST /notes/new (title="Live CLI Note", ...) -> 302 Found /notes/3
GET / -> 200 ("Live CLI Note" present, "Log out" p
```

VERIFIED DIRECTLY — THE REAL SERVER-SIDE LOG FOR THIS EXACT SEQUENCE

`POST /login` and `POST /notes/new` both logged a genuine `302` server-side; the client-observed status of `200` for each is `urllib`'s own default opener transparently following the redirect and reporting the final page's status instead — the identical real distinction Chapter 3 already verified two ways (a following opener vs. a non-following one) for `Response.redirect()` in isolation, now showing up again inside a real multi-step flow.

A Real XSS Attempt, Escaped on the Page, Unescaped (Correctly) in the API

A real note created with `title = "<script>alert(1)</script>"`, while authenticated, through the live server:

```
GET / -> body contains "&lt;script&gt;alert(1)&lt;/script&gt;" (escaped)
      -> body does NOT contain a literal "<script>alert(1)</script>"
```

```
GET /api/notes -> b'[{ "id": 3, "title": "<script>alert(1)</script>", "body": "..."}]'
-> the raw, unescaped tag, correctly, inside a JSON string
```

VERIFIED DIRECTLY — TWO GENUINELY DIFFERENT, BOTH GENUINELY CORRECT OUTCOMES

The HTML page correctly shows the escaped form — `note_card.html`'s own `{{ title }}` ran it

`Response.json()` calls `json.dumps()` directly, with no HTML escaping applied at all, because HTML escaping is meaningless (and would be a real bug) inside a JSON string a client is going to parse as data, not render as markup. Both are correct; each is correct for a genuinely different reason, tied to what actually consumes the output.

A Real SQL Injection Attempt Against the Search Feature

`Note.search()` is new code this chapter wrote, but it reuses Chapter 7's own exact parameterized-query discipline rather than reintroducing the f-string mistake that chapter's own `find_author_naive()` made:

```
@classmethod
def search(cls, conn, query):
    sql = "SELECT id, title, body, author_id FROM notes WHERE title LIKE ? ORDER BY"
    rows = conn.execute(sql, (f'%{query}%',)).fetchall()
    return [cls(**dict(zip(cls._fields.keys(), row))) for row in rows]
```

VERIFIED DIRECTLY — THE REAL, EXECUTED SQL, CHECKED AGAINST SQLITE'S OWN TRACE LOG

A real `GET /?q=x' OR '1'='1` against the live app returns zero notes — not both real notes leaked, which is what the naive version from Chapter 7 would have produced. Checked directly against SQLite's own `set_trace_callback()`, the actual query sent to the database is `...WHERE title LIKE '%x' OR '1'='1%'...` — the driver's own real escaping of the embedded quotes, sent as one bound parameter, with the entire malicious string treated as a single literal search term nothing in the real data matches.

Static Files & Path Traversal — Still Blocked, in the Full App

```
GET /static/style.css          -> 200  Content-type: text/css
GET /static/../manage.py      -> 404  Not Found
```

VERIFIED DIRECTLY — CHAPTER 9'S OWN FIX, UNCHANGED, STILL HOLDING INSIDE A REAL FULL APP

`safe_static_path()` and `serve_static()` were wired in with zero modification from Chapter 9. The legitimate stylesheet request succeeds with the correct real `Content-Type`; a real attempt to escape `STATIC_DIR` and read this app's own `manage.py` source file returns a plain `404`, exactly as verified in isolation.

The Real Payoff: A Genuine Restart, Session and Data Both Intact

This is the concrete, end-to-end version of what Chapter 7 promised the moment it built `SQLiteSessionStore`. Server 1 is killed outright — not gracefully shut down, a real `taskkill /F` — and a genuinely fresh server 2 is started, on a *different port*, sharing nothing with server 1 except the same real file on disk, `notes.db`:

```
# server 1, port 8971: log in, create "Live CLI Note", confirm both are visible
# server 1 is then killed outright (taskkill /F) -- simulating the whole process ending

# server 2: a brand-new App, a brand-new SQLiteSessionStore, PORT 8972,
# hit with the SAME real cookie jar the client has held the whole time
GET / on server 2 -> 200
still logged in (a real "Log out" link present):      True
the note created on server 1 is still there:          True
```

VERIFIED DIRECTLY, END TO END — NOTHING ABOUT THIS APP REMEMBERS ANYTHING EXCEPT THE REAL DATABASE FILE

Both real facts survive a genuine process death and a fresh process, on a different port, with a completely new `App` instance: the logged-in session (because `SQLiteSessionStore` reads it back from the real `sessions` table, keyed by the same session ID the client's cookie jar kept sending) and the note itself (because `Note.save()` committed it to the same real `notes` table). Nothing about the client changed at all — it never even noticed the server underneath it was replaced.

A final, lighter confirmation closes the loop on Chapters 8-9 directly: spawned with `--debug`, a real edit to `manage.py`'s own source while the server keeps running triggers a genuine `os.execv()` reload — the PID listening on the port changed from a real `10560` to a real `36420`, and a request made immediately afterward still returned `200`. The exact same Windows-specific finding from Chapters 8 and 9, reproduced a third time, on this exact app.

Chapter Attribution

PIECE OF NOTES	CHAPTER	WHAT IT REUSED, UNMODIFIED
Every route's own dispatch	2	<code>Router</code> , typed converters, named routes (<code>url_for('note_detail', id=...)</code>)
Every handler's own request/response	3	<code>Request</code> , <code>Response.html().json().redirect()</code>
Every rendered page	4	<code>TemplateEngine</code> , <code>extends</code> / <code>block</code> , auto-escaping, <code>SafeString</code>
<code>App</code> itself, the middleware chain	5	<code>App.use()</code> , onion-model composition, <code>request.state</code>
Login, logout, "who's viewing this page"	6	<code>SessionStore</code> interface, cookie handling, <code>HttpOnly</code>
Every note, every user, the session table itself	7	<code>Field</code> / <code>Model</code> , parameterized queries, <code>SqliteSessionStore</code>
The outermost middleware, <code>--debug</code>	8	<code>make_debug_middleware()</code> , <code>run_with_reloader()</code> , <code>FileWatcher</code>
<code>/static/...</code> , the CLI itself	9	<code>safe_static_path()</code> , <code>serve_static()</code> , <code>argparse</code> -based <code>main()</code>
<code>login_required</code> , <code>Note.search()</code> , the SafeString note-card trick	10 (this chapter)	Genuinely new — built entirely from the eight rows above, with zero new framework code

Course Complete

Ten chapters, one real, working framework: a WSGI-compliant routing engine with typed converters; ergonomic request/response objects; a real template engine with auto-escaping

and inheritance; onion-model middleware; cookie-backed sessions with a genuine persistence option; a minimal, descriptor-based ORM; a real debug page and a live-reload dev server; real static file serving with a genuine path-traversal fix; and a small CLI tying all of it together. Every class, every fix, and every documented finding along the way — the WSGI double-read hang, the naming collision in `render()`, the missing `next_()` crash, the SQL injection, the N+1 query, SQLite's own thread restriction, and the Windows-specific `os.execv()` PID change — was verified directly against real, running code, not asserted. This chapter's own app is the proof: every one of those pieces, together, for the first time, running a real, working site.

Hands-On Exercises

EXERCISE 1

Add a real POST `/notes/<int:id>/delete` route to this chapter's own app, wrapped in `login_required`, that deletes the note's own row from the real notes table and redirects home — then verify, via a real authenticated request and a real unauthenticated one against the same route, that the authenticated request genuinely removes the row (confirmed by a direct row-count check and a real 404 on the deleted note's own detail page afterward) while the unauthenticated one leaves the row count completely unchanged.

 [View solution](#)

EXERCISE 2

Construct a real SQL injection attempt against this chapter's own live search feature (a real GET `?q=...` request with a crafted value), verify the response shows zero notes matched rather than every real note leaking, and then confirm directly — using `sqlite3`'s own `set_trace_callback()` — exactly what real SQL string was sent to the database for that request.

 [View solution](#)

EXERCISE 3

Send a real, unauthenticated POST to `/notes/new` (no session cookie at all) and verify two things directly: the response is a real redirect to `/login`, and — checked against the real notes table's own row count before and after — that `new_note_post()` genuinely never ran at all, rather than running and merely hiding its own result.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE — THE WHOLE COURSE

- **Ch.2 Router** — typed converters, registration order, 404 vs. 405, named/reverse routing
- **Ch.3 Request/Response** — lazy query/body parsing, the double-read-hangs-a-live-socket finding, automatic Content-Length
- **Ch.4 TemplateEngine** — auto-escaping/SafeString, extends/block inheritance, compile-once caching
- **Ch.5 App & middleware** — the onion model, short-circuiting, request.state, the real crash from a forgotten `next_()`
- **Ch.6 Sessions & cookies** — why sessions need cookies at all, server-side expiry as an independent control, `HttpOnly`
- **Ch.7 The ORM** — Field/Model, real SQL injection closed by parameterization, the N+1 fix, `SqliteSessionStore`
- **Ch.8 Debug & live reload** — a real traceback page, `FileWatcher`, `os.execv()`-based restart
- **Ch.9 Static files & the CLI** — the path converter's own dual nature, `safe_static_path()`, `argparse`'s `runserver` command
- **Ch.10 This chapter** — all of the above, wired into one real, working app, verified live through the actual CLI, including a genuine process restart that loses nothing