Server-Side Web Programming

Databases

Pierre Senellart

UNIVERSITÉ PARIS

01 June 2026

Outline

Where we are

- Two lectures on the client side: HTML for structure and content, CSS for style
- A browser can already display any page we ship to it
- But every page so far is a file on disk: the same for every visitor, the same forever

Where we are

- Two lectures on the client side: HTML for structure and content, CSS for style
- A browser can already display any page we ship to it
- But every page so far is a file on disk: the same for every visitor, the same forever

To put a database behind a Web page we need a program that

- runs every time a URL is requested
- reads from (and writes to) the database
- generates HTML on the fly from the data
- collects user input from forms and acts on it

Static vs dynamic

static site each URL is a file on disk; the Web server (nginx, Apache…) just reads it and ships it. Fast and simple, but blind to data.

dynamic site each URL is handled by code on the server: it inspects the request, runs whatever logic is needed (queries, computations…), and returns an HTML response.

Static vs dynamic

static site each URL is a file on disk; the Web server (nginx, Apache. . .) just reads it and ships it. Fast and simple, but blind to data.

dynamic site each URL is handled by code on the server: it inspects the request, runs whatever logic is needed (queries, computations. . .), and returns an HTML response.

We will build the dynamic kind, in Python, with the Flask micro-framework.

Outline

HTTP
- Motivation
- HTTP, the Protocol of the Web

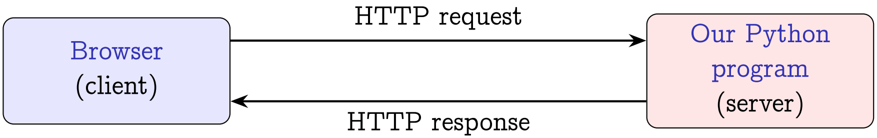
Flask

Requests & responses

State

Wrap-up

Recap: client and server



- The browser and the server speak HTTP (HyperText Transfer Protocol)
- One request, one response; the server has no memory of what came before, unless we add some (cookies, sessions, databases)
- HTTPS = HTTP over TLS: same protocol, encrypted

Underneath HTTP: TCP/IP

HTTP does not reach the network directly; it sits on top of the Internet's protocol stack. Two layers matter to us:

IP (Internet Protocol) delivers individual packets between machines, each identified by an IP address (`93.184.216.34`, or IPv6 `2606:2800:220:1::`). Best-effort: packets may be lost, duplicated or reordered.

TCP (Transmission Control Protocol) adds, on top of IP, a reliable, ordered byte stream between two ports; it handles retransmission, ordering and flow control.

Underneath HTTP: TCP/IP

HTTP does not reach the network directly; it sits on top of the Internet's protocol stack. Two layers matter to us:

IP (Internet Protocol) delivers individual packets between machines, each identified by an IP address (`93.184.216.34`, or IPv6 `2606:2800:220:1::`). Best-effort: packets may be lost, duplicated or reordered.

TCP (Transmission Control Protocol) adds, on top of IP, a reliable, ordered byte stream between two ports; it handles retransmission, ordering and flow control.

The layers, from our text down to the wire:

$$\text{HTTP} \rightarrow \underbrace{\text{TLS}}_{\text{if https}} \rightarrow \text{TCP} \rightarrow \text{IP}$$

A server listens on a port: 80 for HTTP, 443 for HTTPS, 3306 for MySQL...

From name to address: DNS

We type `www.example.com`, but TCP/IP routes to IP addresses. The Domain Name System translates one into the other.

- A huge distributed, hierarchical database of name → address records, run by no single authority
- The browser asks a resolver, which walks the tree: root → `.com` → the server authoritative for `example.com`
- Answers are cached everywhere (with a time-to-live), so most lookups are instant
- Record types: `A` (IPv4), `AAAA` (IPv6), `MX` (mail), `CNAME` (alias)...

From name to address: DNS

We type `www.example.com`, but TCP/IP routes to IP addresses. The Domain Name System translates one into the other.

- A huge distributed, hierarchical database of name → address records, run by no single authority
- The browser asks a resolver, which walks the tree: root → `.com` → the server authoritative for `example.com`
- Answers are cached everywhere (with a time-to-live), so most lookups are instant
- Record types: `A` (IPv4), `AAAA` (IPv6), `MX` (mail), `CNAME` (alias)...

Order of events for one click: resolve the name with DNS → open a TCP connection to that IP on port 443 → TLS handshake → send the HTTP request.

An HTTP request

When you click `https://www.example.com/rooms?city=Paris`, the browser sends, over TCP (and, because of `https`, inside an encrypted TLS layer – more later):

HTTP

```
GET /rooms?city=Paris HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (...)
Accept: text/html
Cookie: session=abc123
```

`GET` the HTTP method (verb)

`/rooms?city=Paris` path + query string

`HTTP/1.1` protocol version

headers `Host`, `Cookie`, `Accept`... metadata, one per line

(POST/PUT requests also carry a body after the headers)

An HTTP response

The server answers:

HTTP

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 173
Set-Cookie: session=abc123; HttpOnly

<!DOCTYPE html>
<html><body><h1>Rooms in Paris</h1>...</body></html>
```

- `200 OK` the status code (see next slide)
- `Content-Type` kind of payload that follows
- blank line separates headers and body
- body the actual HTML (or JSON, image, file. . .)

HTTP methods (verbs)

Two methods carry almost all of the Web, and all of form-based apps:

`GET` retrieve a resource; should have no side effect, safe to repeat, bookmarkable, cacheable

`POST` submit data to create or process a resource; not safe to repeat (think of paying twice)

HTTP methods (verbs)

Two methods carry almost all of the Web, and all of form-based apps:

- `GET` retrieve a resource; should have no side effect, safe to repeat, bookmarkable, cacheable
- `POST` submit data to create or process a resource; not safe to repeat (think of paying twice)

Rule of thumb: `GET` for reads, `POST` for writes. HTML forms only ever generate these two.

HTTP methods (verbs)

Two methods carry almost all of the Web, and all of form-based apps:

- `GET` retrieve a resource; should have no side effect, safe to repeat, bookmarkable, cacheable
- `POST` submit data to create or process a resource; not safe to repeat (think of paying twice)

Rule of thumb: `GET` for reads, `POST` for writes. HTML forms only ever generate these two.

A few other methods exist (`PUT` to replace, `PATCH` to update, `DELETE` to remove), used by programs that talk to a server directly; they need JavaScript or a dedicated client – beyond our scope.

Status codes

Three digits, grouped by first digit:

- 1xx informational (rare)
- 2xx success: `200 OK`, `201 Created`, `204 No Content`
- 3xx redirection: `301 Moved Permanently`, `302 Found`, `303 See Other`, `304 Not Modified`
- 4xx client error: `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `409 Conflict`
- 5xx server error: `500 Internal Server Error`, `503 Service Unavailable`

Status codes

Three digits, grouped by first digit:

- 1xx informational (rare)
- 2xx success: `200 OK`, `201 Created`, `204 No Content`
- 3xx redirection: `301 Moved Permanently`, `302 Found`, `303 See Other`, `304 Not Modified`
- 4xx client error: `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `409 Conflict`
- 5xx server error: `500 Internal Server Error`, `503 Service Unavailable`

Honor them: returning `200 OK` with a page that reads "not found" confuses every search engine, every cache and every accessibility tool on the planet.

HTTPS: HTTP over TLS

TLS (Transport Layer Security, the successor of SSL) is a protocol that wraps an ordinary TCP connection in a secure channel. HTTPS is simply HTTP carried over TLS (default port 443 instead of 80).

HTTPS: HTTP over TLS

TLS (Transport Layer Security, the successor of SSL) is a protocol that wraps an ordinary TCP connection in a secure channel. HTTPS is simply HTTP carried over TLS (default port 443 instead of 80). It provides three guarantees:

confidentiality traffic is encrypted; an eavesdropper on the network sees only ciphertext

integrity any tampering in transit is detected

authentication a certificate proves the server really is `example.com`, not an impostor

HTTPS: HTTP over TLS

TLS (Transport Layer Security, the successor of SSL) is a protocol that wraps an ordinary TCP connection in a secure channel. HTTPS is simply HTTP carried over TLS (default port 443 instead of 80). It provides three guarantees:

confidentiality traffic is encrypted; an eavesdropper on the network sees only ciphertext

integrity any tampering in transit is detected

authentication a certificate proves the server really is `example.com`, not an impostor

- Certificates are signed by a Certificate Authority (CA) the browser trusts; this is the padlock icon
- A TLS handshake negotiates keys before any HTTP is exchanged
- Let's Encrypt issues certificates for free; today the norm is HTTPS everywhere, with plain HTTP redirected to it

HTTP without a browser

HTTP is just text over TCP: we can type a request ourselves and watch the server answer. Plain HTTP (port 80) with `netcat`, ending the request with a blank line:

Console

```
$ nc example.com 80
GET / HTTP/1.1
Host: example.com
Connection: close
```

HTTP without a browser

HTTP is just text over TCP: we can type a request ourselves and watch the server answer. Plain HTTP (port 80) with `netcat`, ending the request with a blank line:

Console

```
$ nc example.com 80
GET / HTTP/1.1
Host: example.com
Connection: close
```

For HTTPS (port 443) the bytes travel inside TLS, so let `openssl` open the encrypted socket first:

Console

```
$ openssl s_client -quiet -crlf -connect example.com:443
GET / HTTP/1.1
Host: example.com
Connection: close
```

HTTP without a browser

HTTP is just text over TCP: we can type a request ourselves and watch the server answer. Plain HTTP (port 80) with `netcat`, ending the request with a blank line:

Console

```
$ nc example.com 80
GET / HTTP/1.1
Host: example.com
Connection: close
```

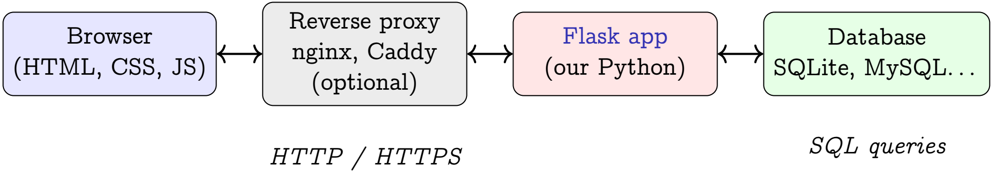
For HTTPS (port 443) the bytes travel inside TLS, so let `openssl` open the encrypted socket first:

Console

```
$ openssl s_client -quiet -crlf -connect example.com:443
GET / HTTP/1.1
Host: example.com
Connection: close
```

Other systems: `nc` and `openssl` ship on macOS too; on Windows use `ncat` (from Nmap) or PowerShell. Everywhere, `curl -v` shows the raw request and response.

What lives where, in our stack



- In development we skip the reverse proxy: Flask itself listens on a port
- In production a real Web server (nginx. . .) sits in front and forwards requests to our app

Outline

Flask

- A micro-framework for Python (2010), built on the Werkzeug toolkit and the Jinja2 template engine
- "Micro": small core, sensible defaults, everything else is a third-party extension
- Free software (BSD-3)
- Documentation: `https://flask.palletsprojects.com/`

Flask

- A micro-framework for Python (2010), built on the Werkzeug toolkit and the Jinja2 template engine
- “Micro”: small core, sensible defaults, everything else is a third-party extension
- Free software (BSD-3)
- Documentation: `https://flask.palletsprojects.com/`

Alternatives in the Python world: Django (full-featured, heavier), FastAPI (modern, API-oriented), Bottle (single file). Pick Flask to start small.

A minimal Flask app

Save as app.py:

Python

```
from flask import Flask

app = Flask(__name__)


@app.route("/")
def index():
    return "<h1>Hello, world!</h1>"
```

A minimal Flask app

Save as app.py:

Python

```
from flask import Flask

app = Flask(__name__)


@app.route("/")
def index():
    return "<h1>Hello, world!</h1>"
```

Run the development server:

Console

```
$ flask --app app run --debug
 * Running on http://127.0.0.1:5000
```

What just happened

- `Flask(__name__)` creates the application object; here `__name__` is `"app"` (the name of app.py), so Flask looks for `templates/` and `static/` next to it
- `@app.route("/")` is a decorator: it registers `index` as the view function for the URL /
- `index` returns a string; Flask wraps it in an HTTP response with `Content-Type: text/html`
- `flask run` starts a small development server that listens on `localhost:5000`
- `--debug` enables auto-reload on file change and a detailed error page in the browser

What just happened

- `Flask(__name__)` creates the application object; here `__name__` is `"app"` (the name of `app.py`), so Flask looks for `templates/` and `static/` next to it
- `@app.route("/")` is a decorator: it registers `index` as the view function for the URL `/`
- `index` returns a string; Flask wraps it in an HTTP response with `Content-Type: text/html`
- `flask run` starts a small development server that listens on `localhost:5000`
- `--debug` enables auto-reload on file change and a detailed error page in the browser

Important: the dev server is not for production use; a real deployment runs your app behind a faster, hardened production server.

Outline

Several routes

Python

```python
@app.route("/")
def index():
    return "<h1>Welcome to our hotel</h1>"


@app.route("/about")
def about():
    return "<p>Family-run since 1923.</p>"


@app.route("/contact")
def contact():
    return "<p>Email: info@hotel.example</p>"
```

- One view function per URL
- The function name is irrelevant to the URL; only the decorator path matters
- URL routing is done by Werkzeug's matcher: first match wins

Variable parts in a URL

Capture a piece of the URL and pass it to the view:

Python

```
@app.route("/rooms/<int:room_id>")
def show_room(room_id):
    return f"<h1>Room {room_id}</h1>"
```

Available converters:

- `string` (default) any text without a slash
- `int` non-negative integer; passed as `int`
- `float` floating-point number
- `path` like `string` but accepts slashes

Variable parts in a URL

Capture a piece of the URL and pass it to the view:

Python

```
@app.route("/rooms/<int:room_id>")
def show_room(room_id):
    return f"<h1>Room {room_id}</h1>"
```

Available converters:

- `string` (default) any text without a slash
- `int` non-negative integer; passed as `int`
- `float` floating-point number
- `path` like `string` but accepts slashes

Visiting `/rooms/504` calls `show_room(504)`; visiting `/rooms/abc` returns `404 Not Found`.

Multiple URLs, one view

Python

```python
@app.route("/")
@app.route("/home")
def index():
    return "<h1>Welcome</h1>"
```

And the converse: one URL, several methods (see next section).

Building URLs the right way

Hard-coding URLs in your templates is fragile; use url_for:

Python

```
from flask import url_for


@app.route("/")
def index():
    return f'<a href="{url_for("show_room", room_id=504)}">Room
    ↪  504</a>'
```

- Takes the view function name and any URL variables
- Returns the correct URL, even if the route is later changed
- Works the same inside Jinja templates:
  `{{ url_for('show_room', room_id=504) }}`

Outline

The query string: request.args

For /search?q=alice&city=Paris:

Python

```python
from flask import request


@app.route("/search")
def search():
    q    = request.args.get("q", "")
    city = request.args.get("city", "")
    return f"<p>Looking for '{q}' in {city}</p>"
```

`request.args` a dict-like object with the query-string parameters

`.get(key, default)` returns the default if the key is missing, no `KeyError`

`.getlist(key)` for repeated keys (`?bed=single&bed=double`)

Form submissions: request.form

Recall from the HTML lecture:

HTML

```html
<form action="/book" method="post">
  <input type="text" name="name">
  <input type="number" name="nights">
  <button type="submit">Book</button>
</form>
```

Python

```python
@app.route("/book", methods=["POST"])
def book():
    name   = request.form["name"]
    nights = int(request.form.get("nights", 1))
    return f"<p>Booking for {name}, {nights} nights</p>"
```

Form submissions: `request.form`

Recall from the HTML lecture:

HTML

```html
<form action="/book" method="post">
  <input type="text" name="name">
  <input type="number" name="nights">
  <button type="submit">Book</button>
</form>
```

Python

```python
@app.route("/book", methods=["POST"])
def book():
    name   = request.form["name"]
    nights = int(request.form.get("nights", 1))
    return f"<p>Booking for {name}, {nights} nights</p>"
```

Where it goes: `request.form` for POST bodies, `request.args` for the query string, regardless of method.

One route, several methods

Same URL, different behavior for GET (show the form) and POST (process it):

Python

```python
@app.route("/book", methods=["GET", "POST"])
def book():
    if request.method == "POST":
        name = request.form["name"]
        # ... insert in the database ...
        return f"<p>Thanks {name}, you are booked.</p>"
    return '''
      <form method="post">
        <input name="name"> <button type="submit">Book</button>
      </form>'''
```

One route, several methods

Same URL, different behavior for GET (show the form) and POST (process it):

Python

```
@app.route("/book", methods=["GET", "POST"])
def book():
    if request.method == "POST":
        name = request.form["name"]
        # ... insert in the database ...
        return f"<p>Thanks {name}, you are booked.</p>"
    return '''
      <form method="post">
        <input name="name"> <button type="submit">Book</button>
      </form>'''
```

Idiomatic pattern: GET renders the empty form, POST validates and acts. We will see a cleaner version in a moment.

File uploads

An HTML form with enctype="multipart/form-data" can carry files:

HTML

```html
<form method="post" enctype="multipart/form-data">
  <input type="file" name="photo">
  <button type="submit">Upload</button>
</form>
```

Python

```python
from werkzeug.utils import secure_filename


@app.route("/upload", methods=["POST"])
def upload():
    f = request.files["photo"]
    f.save(f"uploads/{secure_filename(f.filename)}")
    return "Uploaded."
```

File uploads

An HTML form with enctype="multipart/form-data" can carry files:

HTML

```html
<form method="post" enctype="multipart/form-data">
  <input type="file" name="photo">
  <button type="submit">Upload</button>
</form>
```

Python

```python
from werkzeug.utils import secure_filename


@app.route("/upload", methods=["POST"])
def upload():
    f = request.files["photo"]
    f.save(f"uploads/{secure_filename(f.filename)}")
    return "Uploaded."
```

Always pass the user-supplied filename through secure_filename before writing to disk: never trust client input as a path.

Headers, cookies, environment

`request` also exposes:

`request.method` GET, POST...

`request.path` the path part of the URL

`request.url` the full URL

`request.headers` dict of HTTP headers

`request.cookies` dict of cookies sent by the browser

`request.remote_addr` client IP (often the reverse proxy's, in production)

`request.json` parsed JSON body (when `Content-Type` is `application/json`)

Outline

HTTP

Flask

Requests & responses

- Query String and Forms
- Templates with Jinja
- Static Files and Redirects

State

Wrap-up

The problem with HTML-in-strings

Returning HTML from a Python string works for one paragraph; it breaks down quickly:

- escaping (<, >, &, ") easy to forget → Cross-Site Scripting (XSS) bugs: user input runs as code in the browser
- Python and HTML mixed in one file → unreadable
- no way for a designer to edit the page
- every loop and condition rewritten by hand

The problem with HTML-in-strings

Returning HTML from a Python string works for one paragraph; it breaks down quickly:

- escaping (<, >, &, ") easy to forget → Cross-Site Scripting (XSS) bugs: user input runs as code in the browser
- Python and HTML mixed in one file → unreadable
- no way for a designer to edit the page
- every loop and condition rewritten by hand

Solution: keep HTML in template files, with small placeholders the engine replaces with data. Flask ships with Jinja2.

A first template

templates/room.html:

Jinja

```
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Room {{ room.id }}</title></head>
  <body>
    <h1>Room {{ room.id }}</h1>
    <p>Capacity: {{ room.capacity }} guests.</p>
  </body>
</html>
```

In the view:

Python

```
from flask import render_template


@app.route("/rooms/<int:room_id>")
def show_room(room_id):
    room = ...  # fetched from the database, next lecture
    return render_template("room.html", room=room)
```

Jinja syntax

Three kinds of markers inside a template:

`{{ expression }}` insert the value of an expression, HTML-escaped automatically

`{% statement %}` control flow: `if`, `for`, `set`...

`{# comment #}` ignored, not sent to the browser

Jinja syntax

Three kinds of markers inside a template:

`{{ expression }}` insert the value of an expression, HTML-escaped automatically

`{% statement %}` control flow: `if`, `for`, `set`...

`{# comment #}` ignored, not sent to the browser

Inside expressions you can use most Python operators, attribute access, method calls, and Jinja filters written with a pipe: `{{ name|upper }}`, `{{ price|round(2) }}`.

Loops and conditions

templates/rooms.html:

Jinja

```
<h1>Available rooms</h1>
{% if rooms %}
  <ul>
  {% for room in rooms %}
    <li>Room {{ room.id }}: {{ room.capacity }} guests</li>
  {% endfor %}
  </ul>
{% else %}
  <p>No rooms available.</p>
{% endif %}
```

Every block opener has a matching closer (`{% endfor %}`, `{% endif %}`).

Jinja keywords are not Python

`{% if %}`, `{% for %}`, `{% set %}`... look like Python, but they are Jinja keywords run by the template engine, not Python statements. Watch the differences:

- Every block is closed explicitly (`{% endif %}`, `{% endfor %}`, `{% endblock %}`); indentation carries no meaning, it is just HTML whitespace
- No arbitrary code: assign with `{% set x = 1 %}` (not `x = 1`), and there is no `import` or `def`
- Filters (`name|upper`) and the loop helper `loop.index` are Jinja-only, with no Python equivalent

Jinja keywords are not Python

`{% if %}`, `{% for %}`, `{% set %}`... look like Python, but they are Jinja keywords run by the template engine, not Python statements. Watch the differences:

- Every block is closed explicitly (`{% endif %}`, `{% endfor %}`, `{% endblock %}`); indentation carries no meaning, it is just HTML whitespace
- No arbitrary code: assign with `{% set x = 1 %}` (not `x = 1`), and there is no `import` or `def`
- Filters (`name|upper`) and the loop helper `loop.index` are Jinja-only, with no Python equivalent

The template is rendered to plain HTML on the server; by the time the browser runs the page, no Jinja remains.

Auto-escaping: a built-in defense against XSS

By default, Jinja escapes every value substituted with `{{ }}`. If a guest's name is `<script>alert('xss')</script>`, the rendered HTML is harmless text, not an executable script.

Auto-escaping: a built-in defense against XSS

By default, Jinja escapes every value substituted with `{{ }}`. If a guest's name is `<script>alert('xss')</script>`, the rendered HTML is harmless text, not an executable script. To insert raw HTML deliberately, mark it safe:

Jinja

```
{{ description|safe }}
```

Use sparingly. `|safe` on user-controlled data reopens the very XSS hole auto-escaping closes.

Template inheritance

Most pages of a site share a layout. Define it once, `templates/base.html`:

Jinja

```
<!DOCTYPE html>
<html>
  <head><title>{% block title %}Hotel{% endblock %}</title></head>
  <body>
    <header><h1>Hotel</h1></header>
    <main>{% block content %}{% endblock %}</main>
    <footer>(c) 2026</footer>
  </body>
</html>
```

Extending the base

templates/room.html:

Jinja

```
{% extends "base.html" %}

{% block title %}Room {{ room.id }}{% endblock %}

{% block content %}
  <h2>Room {{ room.id }}</h2>
  <p>Capacity: {{ room.capacity }} guests.</p>
{% endblock %}
```

Extending the base

templates/room.html:

Jinja

```
{% extends "base.html" %}

{% block title %}Room {{ room.id }}{% endblock %}

{% block content %}
  <h2>Room {{ room.id }}</h2>
  <p>Capacity: {{ room.capacity }} guests.</p>
{% endblock %}
```

- Child templates redefine the parent's named blocks
- Layout, navigation and footer live in `base.html` only
- Smaller fragments can be factored with `{% include "partials/card.html" %}`

Outline

Static files

Place CSS, JS, images, fonts under `static/`:

Text
```
myapp/
  app.py
  static/
    style.css
    logo.png
  templates/
    base.html
    rooms.html
```

Flask serves them automatically under `/static/...`. Link them with `url_for`:

Jinja
```
<link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
```

Setting status codes and headers

Return a tuple (body, status, headers):

Python

```
@app.route("/rooms/<int:room_id>")
def show_room(room_id):
    room = lookup(room_id)
    if room is None:
        return render_template("404.html"), 404
    return render_template("room.html", room=room)
```

Or use abort to bail out early:

Python

```
from flask import abort

if room is None:
    abort(404)
```

Redirects and the Post/Redirect/Get pattern

After a successful `POST` (booking, login. . .), do not render HTML from the `POST` handler directly: the user pressing reload would resubmit the form.

Redirects and the Post/Redirect/Get pattern

After a successful `POST` (booking, login. . .), do not render HTML from the `POST` handler directly: the user pressing reload would resubmit the form.

Instead, return a `303 See Other` redirect to a `GET` page that shows the result:

Python

```
from flask import redirect, url_for


@app.route("/book", methods=["POST"])
def book():
    # ... insert in the database ...
    return redirect(url_for("booking_confirmed", id=new_id))
```

Post/Redirect/Get (PRG) is the canonical idiom for any state-changing action on the Web.

Flashing messages between requests

After a redirect, how to tell the new page “the booking was saved”?

Python

```python
from flask import flash


@app.route("/book", methods=["POST"])
def book():
    # ...
    flash("Booking confirmed.", "success")
    return redirect(url_for("index"))
```

Jinja

```jinja
{% with messages = get_flashed_messages(with_categories=true) %}
  {% for category, msg in messages %}
    <p class="flash {{ category }}">{{ msg }}</p>
  {% endfor %}
{% endwith %}
```

Outline

HTTP is stateless

Each HTTP request stands on its own; the server, by default, has no way to know that two requests come from the same browser.

HTTP is stateless

Each HTTP request stands on its own; the server, by default, has no way to know that two requests come from the same browser.

To remember who the user is, the server needs help:

cookies tiny key/value pairs the server asks the browser to store, and the browser sends back with every subsequent request to the same site

server-side sessions the server keeps the real data and gives the browser only a random session id in a cookie

Flask sessions

Flask ships a client-side session: the data is stored in a cookie, signed (not encrypted) with a secret key so the browser cannot tamper with it.

Python

```python
from flask import session

app.secret_key = "change-me-in-production"


@app.route("/login", methods=["POST"])
def login():
    session["user"] = request.form["username"]
    return redirect(url_for("dashboard"))


@app.route("/dashboard")
def dashboard():
    user = session.get("user")
    if user is None:
        return redirect(url_for("login_page"))
    return f"<p>Hello, {user}</p>"
```

Limitations and what comes next

- The default client-side session is fine for small data (a username, a language preference) but readable by the user; do not put secrets in it
- For larger data, keep the session server-side: the cookie carries only an id, and the data itself can live in a database – exactly the kind of store this course is about
- Flask proposes extensions that make it easier to implement authentication workflows (passwords, hashing, login, password reset): Flask-Login, Authlib...
- Other security concerns (CSRF, rate limiting, secure headers...) need care too
- All this is out of scope for this course; the goal here is to know what the building blocks are

Outline

Project layout

Text

```
hotel/
  app.py                 # the Flask app
  static/
    style.css
    logo.png
  templates/
    base.html
    index.html
    rooms.html
    room.html
    booking_form.html
```

A complete booking view (without DB yet)

Python

```python
from flask import Flask, render_template, request, redirect, url_for, flash
app = Flask(__name__)
app.secret_key = "demo"
ROOMS = [504, 107, 302]  # fake data; real DB next lecture


@app.route("/")
def index():
    return render_template("index.html", rooms=ROOMS)


@app.route("/book", methods=["GET", "POST"])
def book():
    if request.method == "POST":
        flash(f"Booked room {request.form['room']} "
              f"for {request.form['name']}.", "success")
        return redirect(url_for("index"))
    return render_template("booking_form.html", rooms=ROOMS)
```

What is still missing

- Persistence: rooms and bookings should come from, and be saved to, a database. Next lecture.
- Validation: today the view trusts whatever the form sends. Use `request.form.get(..., type=int)`, `flask-wtf`, or pydantic
- Security: HTTPS, CSRF tokens, rate limiting, authentication
- Production deployment: a production server behind nginx, logs, monitoring

Outline

What server-side gives us

- A way to turn any URL into the result of a Python function
- A standard vocabulary (HTTP methods, status codes, headers) shared by every browser, every framework
- Templates that keep HTML readable and escape user input by default
- Just enough state (cookies, sessions) to recognize users across requests
- A direct path from database row to HTML response – which we will walk along the next lecture

To explore on your own

- Flask documentation and tutorial: `https://flask.palletsprojects.com/`
- MDN HTTP overview: `https://developer.mozilla.org/en-US/docs/Web/HTTP`
- HTTP status code cheat-sheet: `https://httpstatuses.io/`
- Try `curl -v http://...` or your browser's Network tab to see real HTTP traffic
- Run `flask routes` to list all the routes your app registers

License