Web Basics

Python Course · Module 8

ElnurBDa

Module 8 — Web Basics

We connect Python skills to the web stack:

We will go in three stages:

1. **Web pages** — HTML, CSS, JavaScript, and how the browser talks to a server
2. **HTTP & APIs** — requests, responses, status codes, and JSON
3. **Python as a web client/server** — using requests and serving simple pages

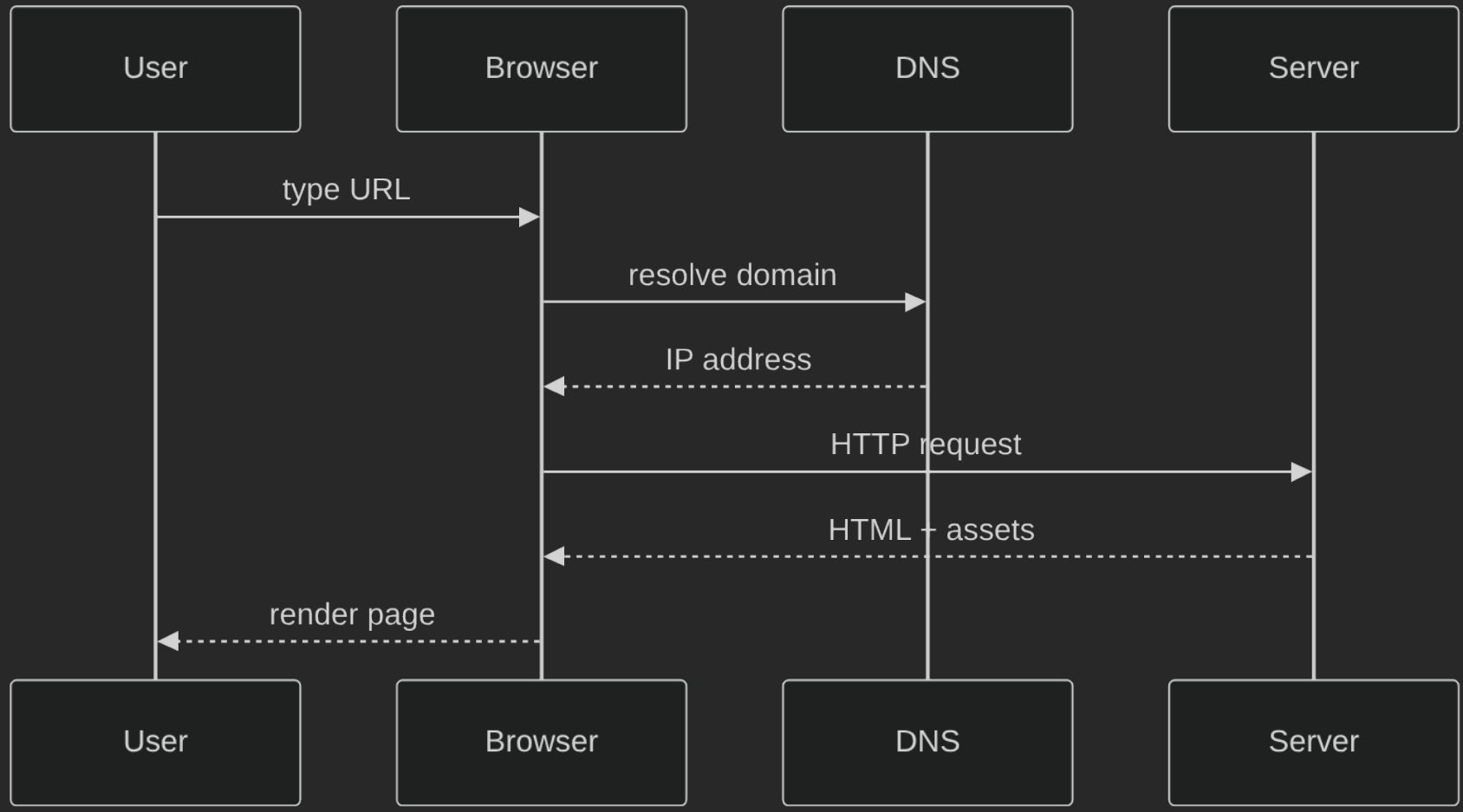
Keep this question in mind:

> “What exactly is traveling over the network right now — HTML or JSON?”

How the Web Works

Step-by-step workflow from browser to server:

1. User enters https://example.com
2. Browser resolves domain → IP
3. Browser sends HTTP request
4. Server responds with HTML, CSS, JS
5. Browser renders page



Section 1 — Web Pages (HTML, CSS, JS)

In this first part we focus on **how a page is built and served**. Do not worry about APIs or JSON yet — just think in terms of:

- HTML = structure & content
- CSS = visual style
- JavaScript = interactivity in the browser

Later sections will reuse this when we load data from APIs instead of hard-coding it.

HTML — Building Blocks

HTML describes structure/content of web pages (like blueprints of a house).

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Hello Web</title>
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is my first web page.</p>
    <a href="https://python.org">Python.org</a>
  </body>
</html>
```

- Tags usually come in pairs: `<tag>content</tag>`
- Attributes add metadata: `<a href="https://python.org">`
- Browser builds the **DOM tree** from these nested tags

HTML Storyboard: Code → Server → Browser

1. **Write HTML** (index.html)
2. **Serve it** (python3 -m http.server 8000)
3. **View in browser** (http://localhost:8000)

```
[index.html] --(http.server)--> [localhost:8000] --(render)--> [Page]
```

Focus questions:

- Do tags nest correctly? (<main> contains <section>)
- Did you include alt text for images?
- Can you describe each section aloud?

HTML Document Anatomy

Section	Purpose
`<!DOCTYPE>`	Tells browser to use modern HTML5 rules
`<html>`	Root element wrapping everything
`<head>`	Metadata, title, linked CSS/JS
`<body>`	Visible content: text, images, forms

Example with more tags:

```
<body>
  <header><h1>Cyber Café</h1></header>
  <main>
    <section>
      <h2>Menu</h2>
      <ul>
        <li><strong>Espresso</strong> — 3 AZN</li>
        <li><strong>Tea</strong> — 2 AZN</li>
      </ul>
    </section>
    <img src="assets/cafe.jpg" alt="Cafe interior" />
  </main>
  <footer>&copy; 2025 Cyber Café</footer>
</body>
```

Core HTML Tags & Patterns

Tag	What it does	Example snippet
`<h1>`	Page title	`<h1>News</h1>`
`<p>`	Paragraph	`<p>Breaking updates...</p>`
`<a>`	Link to another page/file	`<a href="/about.html">About</a>`
`<img>`	Display image	`<img src="logo.png" alt="Logo" />`
`<ul>/<ol>`	Lists (unordered/ordered)	`<ul><li>Item</li></ul>`
`<form>`	User input	`<form><input name="email" /></form>`
`<div>`/`<span>`	Layout/grouping	`<div class="card">...</div>`

Tip: Start simple, add CSS later to style it.

Styling with CSS

CSS (Cascading Style Sheets) controls presentation.

Ways to include CSS:

- Inline: `<p style="color:red">`
- `<style>` block in `<head>`
- External file: `<link rel="stylesheet" href="styles.css" />`

Example:

```html
<head>
  <link rel="stylesheet" href="assets/styles.css" />
</head>
<body>
  <h1 class="hero">Welcome</h1>
</body>
```

```css
.hero {
  color: #ff9900;
  text-align: center;
  border-bottom: 2px solid #444;
}
```

Guideline: keep structure (HTML) separate from style (CSS) for clarity.

JavaScript in HTML

JavaScript brings interactivity.

Common patterns:

- Inline handlers: `<button onclick="alert('Hi')">`
- `<script>` tag inside HTML file
- External file: `<script src="app.js"></script>`

Example:

```
<body>
  <p id="greeting">Loading...</p>
  <button id="btn">Say hi</button>

  <script>
    const message = "Hello from JS!";
    document.getElementById("btn").addEventListener("click", () => {
      document.getElementById("greeting").textContent = message;
    });
  </script>
</body>
```

Place scripts near the end of `<body>` so HTML loads before JS runs.

Serving index.html Quickly

Browsers look for index.html by default.

```
cd website_project
python3 -m http.server 8000
```

- Place index.html alongside the command
- Visit http://localhost:8000 to see it rendered
- Great for checking static prototypes before moving to frameworks

Challenge: create index.html with header, paragraph, list, and image, then serve it locally.

Section 2 — HTTP & APIs

Now that you know what a **page** looks like, we zoom in on the **conversation** between browser and server.

Questions to keep in mind:

- Who is the **client** in this example? Who is the **server**?
- What is the **URL**, what is the **method** (GET/POST/...), and what is inside the **body** (if any)?
- Is the response returning **HTML** or **JSON**?

Once this feels comfortable, we will replace “browser” with **Python code** using `requests`.

HTTP — The Conversation Rules

HTTP (HyperText Transfer Protocol) defines how clients & servers talk — like a script both sides follow.

Structure:

```
<METHOD> <PATH> HTTP/1.1
Header: Value
Header: Value

<optional body>
```

Request example:

```
GET /api/users HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: application/json
```

Response example:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "ok", "count": 2}
```

Common verbs:

- GET (read), POST (create), PUT/PATCH (update), DELETE (remove)

HTTP Anatomy — Request vs Response

Request

```
POST /login HTTP/1.1
Host: mysite.com
Content-Type: application/json
Authorization: Bearer <token>

{"email": "user@example.com", "password": "***"}
```

- Method + path
- Headers describe metadata (auth, format)
- Body carries data (optional)

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: session=abc123

{"status": "ok", "message": "Welcome"}
```

- Status line communicates result
- Headers for caching, cookies, etc.
- Body returns JSON/HTML/binary

Status Codes

Code	Meaning	Use case
200	OK	Successful request
201	Created	Resource created (POST)
301	Moved Permanently	Redirect
400	Bad Request	Invalid client data
401	Unauthorized	Missing/invalid auth
404	Not Found	Resource missing
500	Internal Server Error	Server bug/problem

Fun reference with cat photos: https://http.cat

APIs — Digital Waiters

API = Application Programming Interface.

- Think of a restaurant: you (client) tell the waiter (API) what you want, they talk to the kitchen (server), and bring back the dish (data).
- HTTP APIs expose data/actions via URLs, verbs, and JSON payloads.
- Good APIs document their menu (endpoints, parameters, status codes).

Example: Tasks API served from http://localhost:3000

Endpoint	Method	Description
/api/tasks?completed=true	GET	List tasks filtered by completion flag
/api/tasks	POST	Create a new task with title/description flags
/api/tasks/{id}	GET	Fetch one task by its numeric identifier
/api/tasks/{id}	PUT	Update title, description, or completed status
/api/tasks/{id}	DELETE	Remove the task forever
/api/health	GET	Quick health check for the service

Menu analogy:

- Endpoint = dish name
- Query params/body = customization
- Response JSON = meal you receive

JSON — Data Containers

JSON (JavaScript Object Notation) represents structured data with nested objects/lists.

```
{
  "id": 42,
  "username": "admin",
  "scores": [99, 88, 91],
  "profile": {
    "city": "Baku",
    "active": true
  }
}
```

Analogy: dictionaries + lists combined.

Rules of thumb:

- Keys in double quotes, values can be string/number/bool/null/object/array
- Order usually not important, whitespace ignored

Another sample (array of objects):

```
[
  {"task": "learn HTML", "done": true},
  {"task": "build API client", "done": false}
]
```

Analogy: shipping boxes inside a truck — arrays hold boxes, each box (object) has labeled compartments (keys).

From HTTP & JSON to Python Code

Let’s connect the dots:

1. Browser or client sends an **HTTP request** (method + path + headers + body).
2. Server sends back an **HTTP response** with a **status code** and **body**.
3. If `Content-Type: application/json`, the body contains **JSON**, which feels like Python `dict` + `list`.
4. Our Python script with `requests` plays the role of the **client** instead of the browser.

So when you see `response.json()` later, imagine:

> “Take the JSON body from the HTTP response and turn it into Python dictionaries/lists I can work with.”

Inspecting APIs

Tools:

- Browser dev tools → Network tab
- `curl` in terminal:

```
curl -G "http://localhost:3000/api/tasks" \
  --data-urlencode "completed=true"
```

- Postman, Bruno, or Insomnia for graphical testing

Always check: URL, method, headers, body, response code, payload. Tasks API also requires you to pass `completed` as a query parameter when listing.

Python requests Basics

```python
import requests

url = "http://localhost:3000/api/tasks"
params = {"completed": "false"}  # API requires this flag
response = requests.get(url, params=params, timeout=5)

print("Status:", response.status_code)
response.raise_for_status()
tasks = response.json()
print("First task title:", tasks[0]["title"] if tasks else "No tasks yet")
```

snippet +exec is disabled, run with -x to enable

Notes:

- `timeout` prevents hanging forever (network problems)
- `response.status_code` shows the HTTP status (200, 404, 500, …)
- `response.raise_for_status()` throws an exception for 4xx/5xx errors
- `response.text` is the raw string body, `response.json()` parses JSON into Python types

---

Checkpoint — Your First API Call

1. Activate your venv
2. `pip install requests`
3. Run:

```python
import requests

BASE_URL = "http://localhost:3000/api/tasks"
data = requests.get(
    BASE_URL,
    params={"completed": "true"},
    timeout=5,
).json()

print("Total completed tasks:", len(data))
print("Sample record:", data[0] if data else "None")
```

4. Screenshot output for homework

Questions to consider:

- What happens if you forget `.json()`?
- How would you handle an empty list or missing key?

Section 3 — Python as Web Client & Simple Server

In this final part we:

1. Use Python + `requests` as an **HTTP client** talking to the Tasks API.
2. Use `http.server` (and a tiny custom server) as a **simple HTTP server** to serve static HTML.

Key mental model:

- Browser and Python scripts are just **different HTTP clients**.
- Servers (your simple Python server, or the Tasks API on port 3000) all speak the same HTTP "language".

When in doubt, go back to the HTTP request/response examples and map each line to what `requests` or the browser is doing.

Installing requests Inside a venv

```
python3 -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install --upgrade pip
pip install requests
pip freeze
```

- Always isolate dependencies per project
- `pip freeze` helps document versions (`requirements.txt`)
- Deactivate with `deactivate` when done

Sending Data with requests

```python
import requests

payload = {
    "title": "Draft Module 8 slides",
    "description": "Cover Tasks API examples",
    "completed": False,
}

resp = requests.post(
    "http://localhost:3000/api/tasks",
    json=payload,
    timeout=5,
)

resp.raise_for_status()
created = resp.json()
print("New task id:", created["id"])
print("Completed flag:", created["completed"])
```

snippet +exec is disabled, run with -x to enable

`json=...` auto converts dict to JSON & sets `Content-Type: application/json`.

Serving Static HTML with Python

Quick demo using built-in http.server.

```
python3 -m http.server 8000
```

- Serves current directory
- Visit http://localhost:8000 to view files
- Great for testing HTML/CSS locally

Stop with Ctrl+C.

Custom Minimal Server

Serve a specific HTML file with Python.

```python
from http.server import SimpleHTTPRequestHandler, HTTPServer
from pathlib import Path

PORT = 8080
ROOT = Path("website")

class StaticHandler(SimpleHTTPRequestHandler):
    def translate_path(self, path):
        # Map requests into our ROOT directory
        return str(ROOT / path.lstrip("/"))

if __name__ == "__main__":
    ROOT.mkdir(exist_ok=True)
    httpd = HTTPServer(("0.0.0.0", PORT), StaticHandler)
    print(f"Serving on http://localhost:{PORT}")
    httpd.serve_forever()
```

snippet +exec is disabled, run with -x to enable

Place `index.html` inside `website/` and run script.

Explanation:

- `HTTPServer` listens on port 8080 and waits for requests
- `SimpleHTTPRequestHandler` knows how to serve files; we override `translate_path` to force it into `website/`
- `ROOT.mkdir(exist_ok=True)` ensures the folder exists before serving
- Visit http://localhost:8080/index.html to test

Project structure reference:

```
website/
├── index.html
├── about.html
└── assets/
    └── styles.css
server.py
```

Run `python server.py` from the folder containing `website/`.

Mini Project

Build a simple task dashboard backed by the local API:

1. Create `index.html` that renders a table of tasks from `data.json`.
2. Write `fetch_tasks.py` that calls `http://localhost:3000/api/tasks?completed=false` and saves the JSON payload.
3. Add a second button on the page that loads tasks marked as completed (read from another JSON file or switch endpoints dynamically).
4. Serve the folder via `python -m http.server` and demo reloading after rerunning the fetch script.