
Python HTTP Server on Ubuntu

Practical Guide for Setup, Configuration, and Real-World Use

Running a lightweight HTTP server is one of the easiest ways to test, present, or prototype your work on Ubuntu. Python makes this process simple. Its built-in HTTP server provides a fast way to serve files, test APIs, or validate network behavior without heavy web frameworks or external services.

This article walks through how to set up and run a Python HTTP server on Ubuntu, how to secure and customize it, and how to use it in different real-world scenarios.

---

1. Why Use Python's HTTP Server?

Ubuntu already includes tools like Apache or Nginx, so why use Python instead?

The Python HTTP server is ideal when you need:

- **A quick test environment** – Serve a directory over HTTP with a single command. No config files, no services, no packages.
- **A simple way to demo files** – Colleagues can open your work through their browser without SSH or shared drives.
- **A sandbox for API experiments** – You can define a custom handler that responds to GET, POST, PUT, or DELETE requests.
- **A portable tool** – Works the same on Ubuntu, macOS, or Windows. If Python is installed, you're ready.

In short, it's perfect for learning, teaching, prototyping, and troubleshooting.

---

2. Setting Up Your Ubuntu Environment

Most modern Ubuntu distributions include Python 3 by default. Check your version with:

```
python3 --version
```

If Python isn't installed or you want to update it:

```
sudo apt update
sudo apt install python3
```

For development or testing, it's helpful to install:

```
sudo apt install python3-pip
sudo apt install python3-venv
```

These tools let you create virtual environments or extend the simple HTTP server with Python packages.

---

3. Creating a Simple Python HTTP Server

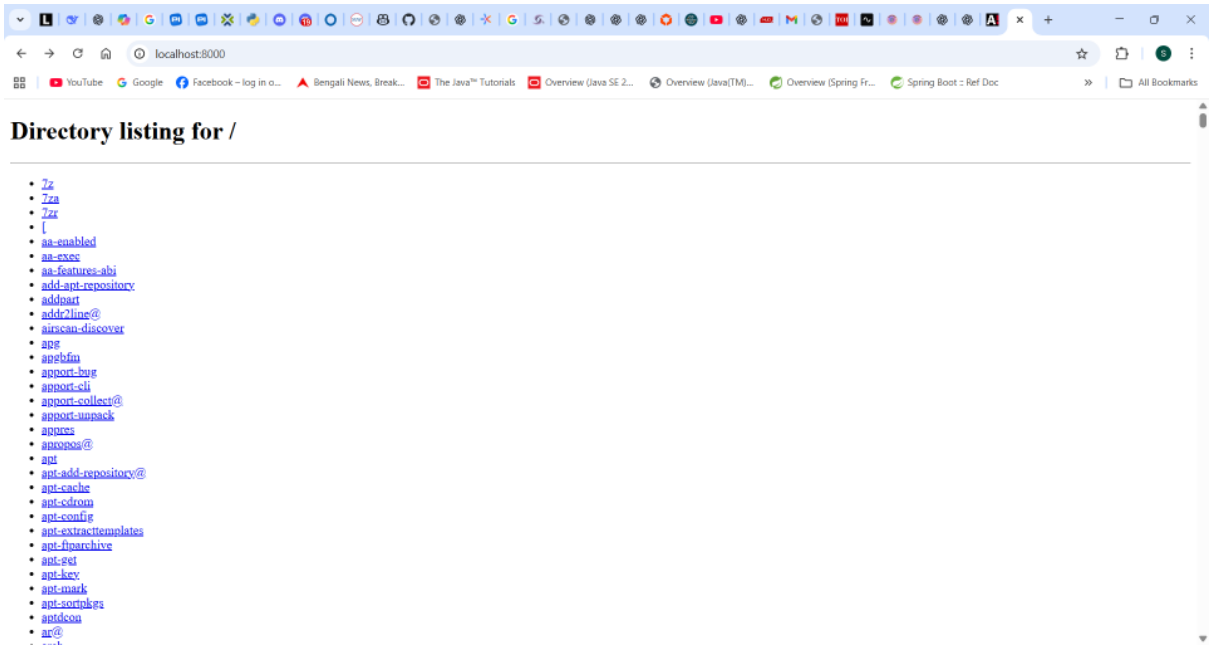
Python 3 includes the `http.server` module to instantly launch a basic web server. Navigate to your desired directory:

```
$ cd /path/to/your/folder
$ python3 -m http.server
```

Http Server will start. Linux Console will show:

```
saigos@LAPTOP-N53D8VJU:/usr/bin$ python3 -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

127.0.0.1 - - [24/Nov/2025 11:28:15] "GET / HTTP/1.1" 200 -

127.0.0.1 - - [24/Nov/2025 11:28:15] code 404, message File not found

127.0.0.1 - - [24/Nov/2025 11:28:15] "GET /favicon.ico HTTP/1.1" 404 -
```

Go to browser, and type: http://localhost:8000. Browser will show:



Since there is no index.html, the page shows a list of files in the directory from which the command was typed.

To use a different port, type:

```
python3 -m http.server 8080
```

To allow access from other devices, type:

```
python3 -m http.server 8000 --bind 0.0.0.0
```

Now the server is live, and anyone who can reach your IP and port can view your directory.

---

4. Managing Firewall Rules with UFW

Ubuntu uses UFW (Uncomplicated Firewall) for firewall management. To enable UFW, type:

```
sudo ufw enable
```

To allow external access, type:

```
sudo ufw allow 8000
```

The command sudo ufw allow 8000 in Ubuntu tells the firewall (UFW) to **open TCP/UDP port 8000** for incoming connections, allowing external clients to reach services running on that port.

Again, type:

```
sudo ufw reload
```

The above command reloads the firewall rules without disabling or re-enabling UFW.

Check status:

```
sudo ufw status
```

For local-only access:

```
python3 -m http.server 8000 --bind 127.0.0.1
```

---

5. Creating a Custom Python HTTP Server

The built-in command is handy, but sometimes you'll want to control responses—like adding headers, handling POST requests, or serving APIs.

Example – Simple API endpoint:

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

class SimpleAPI(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/status":
            self.send_response(200)
            self.send_header("Content-Type", "application/json")
            self.end_headers()
            response = {"server": "running", "path": self.path}
            self.wfile.write(json.dumps(response).encode())
        else:
            self.send_response(404)
            self.end_headers()

def run_server():
    server = HTTPServer(("localhost", 8000), SimpleAPI)
    print("Server running at http://localhost:8000/")
    server.serve_forever()

if __name__ == "__main__":
    run_server()
```

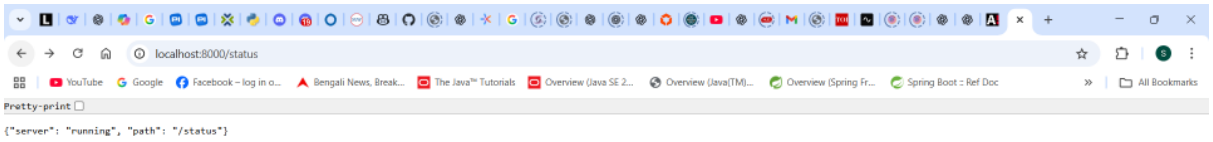
Run it:

```
python3 server.py
```

Test it in your browser:

http://localhost:8000/status

Browser will show:



This structure lets you build mock APIs or test behavior without deploying a full web framework.

6. Running the Server in the Background

To keep the server running long-term, use tools like:

- `systemd`
- `tmux`
- `screen`
- `nohup`

Example with `nohup`:

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

Example with `systemd`:

```
sudo nano /etc/systemd/system/python-http.service
```

Add:

```
[Unit]
Description=Python HTTP Server
After=network.target

[Service]
ExecStart=/usr/bin/python3 -m http.server 8000 --directory /var/www
WorkingDirectory=/var/www
```

```
Restart=always

[Install]
WantedBy=multi-user.target
```

Enable and start:

```
sudo systemctl enable python-http
sudo systemctl start python-http
sudo systemctl status python-http
```

The server will now start automatically on boot.

---

7. Adding HTTPS with a Reverse Proxy

Python's default server doesn't support HTTPS, but you can add it using Nginx as a reverse proxy.

Install Nginx:

```
sudo apt install nginx
```

Configuration example:

```
server {
    listen 80;
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
}
```

Reload Nginx:

```
sudo systemctl reload nginx
```

Add HTTPS with Let's Encrypt:

```
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx
```

Now your Python server is securely accessible via Nginx.

---

8. Troubleshooting Common Issues

- **Address already in use** (in case you have already started http on port 8000, or if there is already another process running on port 8000), type**:**

  ```
  sudo lsof -i :8000
  ```

This command will give process id of program running in port 8000.

Stop the running program using the process id <PID> using the command below:

```
sudo kill <pid>
```

- **Permission denied:** Use accessible directories or adjust permissions.
- **Cannot access from other devices:** Check firewall, bind address, and router isolation.

9. Conclusion

Running a Python HTTP server on Ubuntu is simple yet powerful. With one command, you can share files, test web content, or build temporary APIs. With a bit of customization, it becomes a versatile development tool for any workflow.