How Python Web Development Works

Introduction

Python is not only used for scripts, automation, data science, and AI. It is also widely used to build websites and web applications.

But when you open a website in your browser, Python is usually not running inside the browser. Instead, Python commonly runs on a **server**.

The browser sends a request to the server. The Python application receives that request, performs some work, communicates with databases or other services when necessary, and sends a response back.

A simple mental model is:

```
Browser
   ↓
HTTP Request
   ↓
Web Server
   ↓
Python Web Application
   ↓
Database / APIs / Files
   ↓
Python Web Application
   ↓
HTTP Response
   ↓
Browser
```

Understanding this flow is more important than memorizing individual frameworks.

---

1. What Is Web Development?

Web development means building software that communicates over the web.

A web application usually has several parts:

```
Frontend
    ↓
Backend
    ↓
Database
```

Frontend

The frontend is what the user interacts with.

Common technologies include:

```
HTML
CSS
JavaScript
React
Vue
Angular
```

Backend

The backend handles server-side logic.

Python can be used here.

Examples:

```
Python
Django
Flask
FastAPI
```

Database

A database stores application data.

Examples:

```
PostgreSQL
MySQL
SQLite
MongoDB
```

A complete application might therefore look like:

```
React
   ↓
FastAPI
   ↓
PostgreSQL
```

Python is responsible for the backend in this example.

---

2. Why Is Python Used for Web Development?

Python is useful for web development because it provides:

- readable syntax
- large ecosystem
- mature frameworks
- database support
- authentication libraries
- API development tools
- testing tools
- asynchronous programming support
- strong integration with AI and data systems

Python is especially useful when a web application also needs:

```
AI
Machine Learning
Data Processing
Automation
APIs
Background Jobs
```

For example, an application might receive an uploaded document, process it with Python, send it through an AI model, store the result, and return the result to the user.

---

3. The Most Important Concept: Client and Server

Web applications usually involve two major sides.

Client

The client is usually the user's browser.

Examples:

```
Chrome
Firefox
Safari
Edge
```

The client requests something from a server.

Server

The server is the computer that runs the backend application.

It receives requests and produces responses.

For example:

```
User opens:

https://example.com/products
```

The browser sends a request to the server.

The Python application determines what `/products` means and generates the appropriate response.

---

4. What Happens When You Open a Website?

Suppose you visit:

```
https://example.com/products
```

A simplified flow is:

```
1. Browser receives URL
        ↓
2. DNS finds the server
        ↓
3. Browser connects to server
        ↓
4. Browser sends HTTP request
        ↓
5. Web server receives request
        ↓
6. Python application processes request
        ↓
7. Application may query database
        ↓
8. Python creates response
        ↓
9. Server sends HTTP response
        ↓
10. Browser renders the result
```

There are many details hidden inside this process, but this is the core mental model.

---

5. What Is HTTP?

HTTP stands for:

HyperText Transfer Protocol

It defines how clients and servers communicate.

A browser might send:

```
GET /products HTTP/1.1
Host: example.com
```

The server then sends a response.

For example:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

followed by the response body.

HTTP is the communication layer between the client and server.

---

6. HTTP Methods

HTTP provides different methods for different types of operations.

GET

Used to retrieve data.

```
GET /products
```

Meaning:

> Give me the products.

POST

Used to send or create data.

```
POST /products
```

The request might contain:

```
{
    "name": "Python Course",
    "price": 20
}
```

PUT

Usually used to replace or update a resource.

```
PUT /products/10
```

PATCH

Usually used for a partial update.

```
PATCH /products/10
```

DELETE

Used to remove a resource.

```
DELETE /products/10
```

A useful mental model is:

```
GET     → Read
POST    → Create
PUT     → Replace
PATCH   → Partially update
DELETE  → Delete
```

---

7. What Is a Python Web Framework?

Writing an entire web server from scratch would be unnecessary for most applications.

A web framework provides common functionality for building web applications.

Popular Python frameworks include:

Django

A large, batteries-included web framework.

Useful for:

```
Full web applications
Authentication
Admin panels
Database-driven applications
Large projects
```

Flask

A lightweight framework.

Useful when you want more control over application structure.

FastAPI

A modern framework particularly popular for APIs.

It provides:

```
API routing
Request validation
Type-hint based schemas
Automatic API documentation
Async support
```

The framework handles much of the repetitive infrastructure so developers can focus on application logic.

---

8. A Simple Flask Application

A minimal Flask application can look like this:

```
from flask import Flask

app = Flask(__name__)


@app.route("/")
def home():
    return "Hello from Python!"


if __name__ == "__main__":
    app.run()
```

The important part is:

```
@app.route("/")
def home():
    return "Hello from Python!"
```

This tells Flask:

> When someone requests `/`, run the `home()` function.

The function returns the response.

---

9. A Simple FastAPI Application

FastAPI provides another approach:

```
from fastapi import FastAPI
```

```
app = FastAPI()


@app.get("/")
def home():
    return {"message": "Hello from Python!"}
```

Now a request to:

```
GET /
```

can produce:

```
{
    "message": "Hello from Python!"
}
```

This is especially useful for APIs.

---

10. What Is Routing?

Routing means deciding which piece of code should handle a particular URL.

For example:

```
/                  → home
/products          → products
/products/10       → product_detail
/about             → about
/contact            → contact
```

In Flask:

```
@app.route("/products")
def products():
    return "Products"
```

The framework connects the URL to the Python function.

This relationship is:

```
URL
 ↓
Route
 ↓
Python Function
```

```
 ↓
Response
```

---

11. Dynamic Routes

Routes can contain values.

For example:

```
/products/25
```

The `25` might represent a product ID.

Flask:

```
@app.route("/products/<int:product_id>")
def product(product_id):
    return f"Product ID: {product_id}"
```

Request:

```
/products/25
```

Result:

```
Product ID: 25
```

The value from the URL becomes a Python variable.

---

12. What Is a Request?

A request contains information sent by the client.

It can include:

```
HTTP method
URL
Headers
Query parameters
Path parameters
Cookies
Request body
```

For example:

```
GET /products?page=2
```

Here:

```
/products
```

is the path.

```
page=2
```

is a query parameter.

---

13. Query Parameters

Query parameters are values included after `?`.

Example:

```
/products?category=books&page=2
```

There are two parameters:

```
category = books
page = 2
```

They are commonly used for:

```
Filtering
Searching
Pagination
Sorting
```

---

14. Request Body

When sending data to a server, the client can include a request body.

For example:

```
{
    "username": "hafsa",
    "email": "user@example.com"
}
```

A Python application can read and validate this data.

FastAPI makes this particularly convenient.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
```

```
class User(BaseModel):
    username: str
    email: str


@app.post("/users")
def create_user(user: User):
    return user
```

FastAPI uses the model to validate incoming data.

---

15. What Is a Response?

A response is what the server sends back to the client.

It can contain:

```
HTML
JSON
Text
Files
Images
Status codes
Headers
```

For an API, JSON is extremely common.

Example:

```
{
    "id": 10,
    "name": "Python"
}
```

For a traditional website, the response might contain HTML.

---

16. HTTP Status Codes

The server also communicates the result through a status code.

Common codes include:

```
200 → Success
201 → Created
204 → Success with no response body
```

```
400 → Bad Request
401 → Authentication required
403 → Forbidden
404 → Not Found
500 → Server Error
```

For example:

```
GET /products/10
```

If product 10 exists:

```
200 OK
```

If it doesn't:

```
404 Not Found
```

Status codes allow clients to understand what happened without interpreting arbitrary text.

---

17. Python and HTML

Python can generate HTML responses.

For example, a Flask application can render a template:

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route("/")
def home():
    return render_template("index.html")
```

The HTML file might contain:

```
<!DOCTYPE html>
<html>
<head>
    <title>Python App</title>
</head>
<body>
    <h1>Hello from Python</h1>
</body>
</html>
```

The browser receives HTML and renders it.

---

18. Template Engines

Instead of manually creating HTML strings in Python, frameworks can use templates.

For example:

```
<h1>Welcome, {{ username }}</h1>
```

Python can provide:

```
return render_template(
    "profile.html",
    username="Hafsa"
)
```

The template engine combines the data with the HTML.

Result:

```
<h1>Welcome, Hafsa</h1>
```

This is useful for server-rendered applications.

---

19. Python and Databases

Most useful web applications need persistent data.

For example:

```
Users
Products
Orders
Articles
Comments
Resources
```

Python can communicate with databases using database drivers, ORMs, and other libraries.

A simplified flow is:

```
HTTP Request
     ↓
Python Route
     ↓
Business Logic
     ↓
```

```
Database Query
     ↓
Database
     ↓
Python
     ↓
HTTP Response
```

---

20. What Is an ORM?

ORM stands for:

Object Relational Mapper

An ORM allows developers to work with database records using programming-language objects instead of writing every SQL statement manually.

For example, conceptually:

```
user = User(
    name="Hafsa",
    email="user@example.com"
)
```

The ORM can translate this into database operations.

Popular Python ORM options include:

```
Django ORM
SQLAlchemy
```

ORMs can improve developer productivity, but understanding SQL remains important.

---

21. Python Web Applications and APIs

A web application does not always return HTML.

It can provide an API.

For example:

```
GET /api/resources
```

could return:

```
[
    {
        "id": 1,
```

```
        "title": "Python Basics"
    },
    {
        "id": 2,
        "title": "OOP in Python"
    }
]
```

A frontend application can consume this API.

The architecture might be:

```
React
   ↓
HTTP Request
   ↓
FastAPI
   ↓
Python Logic
   ↓
PostgreSQL
   ↓
FastAPI
   ↓
JSON
   ↓
React
```

---

22. Traditional Website vs API Backend

There are two common patterns.

Server-rendered application

```
Browser
   ↓
Python
   ↓
HTML
   ↓
Browser
```

The Python application generates HTML.

API-based application

```
Browser / Mobile App
        ↓
      API
        ↓
     Python
        ↓
    Database
        ↓
      JSON
```

The frontend separately handles presentation.

Neither architecture is automatically appropriate for every application.

---

23. Authentication

Many web applications need to know who the user is.

Examples:

```
Login
Logout
Registration
Password reset
Sessions
Tokens
```

A simplified login flow:

```
User enters email + password
          ↓
Browser sends request
          ↓
Python backend
          ↓
Check stored password hash
          ↓
Authentication succeeds
          ↓
Session / token created
          ↓
User accesses protected resources
```

Passwords should not be stored as plain text.

A backend should use appropriate password hashing mechanisms.

---

24. Cookies and Sessions

A browser can store cookies associated with a website.

A cookie might contain a session identifier.

Simplified flow:

```
Login
 ↓
Server creates session
 ↓
Server sends session cookie
 ↓
Browser stores cookie
 ↓
Future requests include cookie
 ↓
Server identifies session
```

Cookies can also be used for other purposes, but authentication-related cookies require careful security configuration.

---

25. Authentication vs Authorization

These are different concepts.

Authentication

Answers:

> Who are you?

Authorization

Answers:

> What are you allowed to do?

For example:

```
Authentication:
Hafsa is logged in.

Authorization:
```

```
Hafsa can edit her own resource.
Hafsa cannot access another user's private settings.
```

A secure application needs both when protected functionality is involved.

---

26. Middleware

Middleware is code that runs between the incoming request and the final request handler.

Conceptually:

```
Request
   ↓
Middleware
   ↓
Authentication
   ↓
Logging
   ↓
Route
   ↓
Response
```

Middleware can be used for tasks such as:

```
Authentication
Logging
CORS
Request processing
Security headers
Timing
Error handling
```

---

27. What Is CORS?

CORS stands for:

Cross-Origin Resource Sharing

Suppose:

```
Frontend:
https://app.example.com

API:
https://api.example.com
```

The browser treats these as different origins.

The API must provide appropriate CORS rules if the browser should allow the frontend to access it.

CORS is primarily a browser security mechanism.

It is not an authentication system.

---

28. Static Files

Web applications often need static assets:

```
CSS
JavaScript
Images
Fonts
Icons
```

These files usually do not need Python logic for every request.

A production architecture often uses a web server, CDN, or object storage for static assets.

This reduces unnecessary work for the application server.

---

29. Web Server vs Python Application Server

These concepts are often confused.

A web server such as:

```
Nginx
Apache
```

can handle HTTP-related responsibilities.

The Python application server runs the Python application.

For example:

```
Browser
   ↓
Nginx
   ↓
Gunicorn
   ↓
Django / Flask
   ↓
Database
```

For ASGI applications such as FastAPI, servers such as:

```
Uvicorn
```

are commonly used.

The exact production architecture depends on the framework and deployment environment.

---

30. WSGI and ASGI

Python web applications commonly use interfaces that connect application frameworks to servers.

WSGI

WSGI stands for:

Web Server Gateway Interface

It is widely used by traditional synchronous Python web applications.

Examples:

```
Django
Flask
Gunicorn
```

ASGI

ASGI stands for:

Asynchronous Server Gateway Interface

It supports modern asynchronous applications and long-lived connections.

It is commonly used with:

```
FastAPI
Starlette
Uvicorn
```

A simplified distinction:

```
WSGI → Traditional synchronous web applications

ASGI → Async-capable modern web applications
```

This is a simplified mental model; frameworks and servers have additional capabilities.

---

31. Synchronous vs Asynchronous Web Code

Synchronous code generally waits for an operation to finish before continuing.

```
result = get_data()
process(result)
```

Asynchronous code can allow other work to proceed while waiting for certain I/O operations.

```
result = await get_data()
process(result)
```

Async programming can be useful for workloads involving many concurrent I/O operations.

It does not automatically make every application faster.

---

32. Background Tasks

Some work should not block the user's request.

For example:

```
Send email
Generate report
Process large file
Resize images
Run scheduled job
```

Instead of making the user wait:

```
Request
 ↓
Create background job
 ↓
Return response
```

A worker can then process the task.

Common tools in Python ecosystems include:

```
Celery
RQ
Task queues
Cloud-managed job systems
```

---

33. Web Application Architecture

A simple application might start as:

```
Route
 ↓
```

```
Function
 ↓
Database
```

As the application grows, responsibilities can be separated:

```
Routes / Views
      ↓
Services
      ↓
Repositories
      ↓
Database
```

For example:

```
GET /resources
      ↓
Resource route
      ↓
Resource service
      ↓
Resource repository
      ↓
PostgreSQL
```

This makes large applications easier to maintain when the boundaries are designed appropriately.

---

34. Example: haas.dev Resource Request

Imagine a user visits:

```
https://dev-roast-app.vercel.app/resources
```

A simplified architecture could be:

```
Browser
   ↓
Request /resources
   ↓
Web Application
   ↓
Resource Logic
   ↓
```

```
Database / Resource Data
   ↓
Response
   ↓
Browser
```

If the application provides an API:

```
GET /api/resources
```

the response could look like:

```
[
    {
        "title": "Python Security Basics",
        "category": "Python"
    },
    {
        "title": "Debugging Python Applications",
        "category": "Python"
    }
]
```

The frontend could then display these resources.

The important concept is not the exact framework.

The important concept is understanding the flow:

```
Request
→ Routing
→ Application Logic
→ Data
→ Response
```

---

35. Environment Variables

Web applications often need configuration values.

Examples:

```
DATABASE_URL
SECRET_KEY
API_KEY
DEBUG
```

These values should generally not be hard-coded into source code.

Instead:

```
import os

database_url = os.getenv("DATABASE_URL")
```

For local development, a `.env` file may be used with an appropriate library.

Production environments usually provide secrets through their deployment platform or secret-management system.

---

36. Error Handling in Web Applications

Applications must handle failures.

For example:

```
Database unavailable
Invalid input
Missing resource
Expired authentication
Third-party API failure
Unexpected programming error
```

A backend should return appropriate HTTP responses.

For example:

```
Invalid input
→ 400

Not authenticated
→ 401

Not allowed
→ 403

Resource missing
→ 404

Unexpected server failure
→ 500
```

Detailed internal errors should not be exposed to users unnecessarily.

---

37. Validation

Never assume that incoming data is correct.

A client could send:

```
{
    "age": "hello"
}
```

when the application expects:

```
age → integer
```

Validation checks incoming data before it reaches important application logic.

For example:

```
class User(BaseModel):
    name: str
    age: int
```

A validation system can reject invalid data before the application processes it.

Validation improves:

```
Correctness
Security
Reliability
Developer experience
```

---

38. Testing Python Web Applications

Web applications should be tested at multiple levels.

Unit tests

Test small pieces of logic.

```
calculate_price()
validate_email()
```

Integration tests

Test components working together.

```
API + database
```

End-to-end tests

Test a complete user flow.

```
Login
 ↓
Open dashboard
 ↓
Create resource
 ↓
Verify result
```

A framework such as `pytest` can be used across these testing levels.

---

39. Logging

Logging helps developers understand what happened inside the application.

Example:

```
import logging

logger = logging.getLogger(__name__)

logger.info("Resource request received")
```

Production applications should use structured, useful logging rather than random `print()` statements.

Avoid logging secrets, passwords, tokens, and other sensitive information.

---

40. Deployment

Writing the application is only part of web development.

Eventually, the application needs to run somewhere accessible to users.

A simplified deployment flow:

```
Write Python Code
      ↓
Test
      ↓
Commit to Git
      ↓
Build / Install Dependencies
      ↓
Configure Environment
      ↓
Deploy
```

```
      ↓
Application Runs on Server
```

Possible hosting environments include:

```
Cloud platforms
Virtual machines
Containers
Platform-as-a-Service providers
Managed application hosting
```

---

41. Development vs Production

Development:

```
DEBUG = True
Local database
Local environment
Development server
Verbose debugging
```

Production:

```
DEBUG = False
Production database
Secure secrets
Production server
Logging
Monitoring
HTTPS
Backups
```

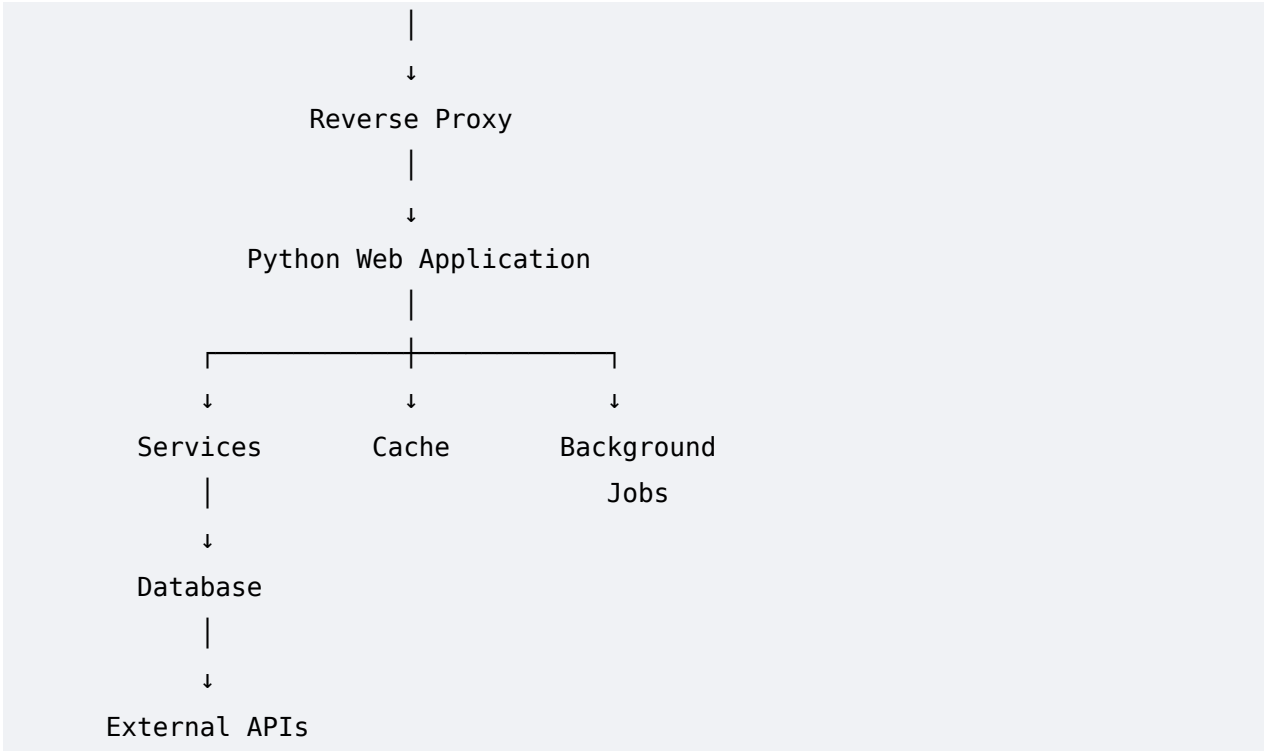
Development configuration should not simply be copied into production.

---

42. A Complete Python Web Architecture

A more realistic architecture might look like:

```
                    Internet
                        |
                        ↓
                    Browser
                        |
                      HTTPS
```



Each component has a responsibility.

The architecture becomes more complex as traffic and requirements grow.

---

43. Common Python Web Development Stack

A Python application might use:

```
Frontend
React

Backend
FastAPI

Database
PostgreSQL

ORM
SQLAlchemy

Testing
pytest

Server
Uvicorn

Reverse Proxy
```

```
Nginx

Version Control
Git

Deployment
Cloud Platform
```

Another project might use:

```
Django
Django ORM
PostgreSQL
Gunicorn
Nginx
```

There is no requirement to use the same stack for every project.

---

44. Common Mistakes

Mistake 1: Thinking Python runs in the browser

Usually, Python runs on the server.

The browser normally executes:

```
HTML
CSS
JavaScript
```

Mistake 2: Thinking a framework is the entire backend

A framework is a tool for building the application.

You still need to understand:

```
HTTP
Routing
Databases
Authentication
Security
Deployment
```

Mistake 3: Putting database logic everywhere

Avoid making every route directly responsible for all database operations.

Separate responsibilities when the application becomes large enough to benefit from it.

Mistake 4: Trusting client input

Everything received from a client should be treated as untrusted input.

Mistake 5: Hard-coding secrets

Never commit API keys, passwords, or production secrets to Git.

Mistake 6: Using async without understanding it

Adding `async` does not automatically improve performance.

Understand the workload first.

Mistake 7: Only testing manually

Manual testing is useful, but automated tests catch regressions much more reliably.

Mistake 8: Ignoring deployment

A developer who understands only local development does not yet understand the complete web application lifecycle.

---

45. How to Think About Python Web Development

Use this mental model:

```
1. Client
   Who is making the request?

2. HTTP
   What is being requested?

3. Routing
   Which code handles it?

4. Validation
   Is the input valid?

5. Business Logic
   What should the application do?

6. Data
   Does it need a database or external API?

7. Response
   What should be returned?

8. Security
```

```
    Is the operation protected?

 9. Testing
    Does it work reliably?

 10. Deployment
     Where does the application run?
```

This framework is useful regardless of whether you use Django, Flask, or FastAPI.

---

46. Django vs Flask vs FastAPI

Feature	**Django**	**Flask**	**FastAPI**
Style	Full framework	Lightweight framework	API-focused modern framework
Built-in features	Many	Minimal	Focused
Templates	Yes	Yes	Possible
ORM	Django ORM	Choose your own	Choose your own
API development	Yes	Yes	Excellent fit
Async support	Supported	Supported in modern versions	Strong focus
Admin interface	Built-in	Extensions	Not built-in
Best fit	Full web application	Flexible smaller/cus	APIs and async-

	s	tom apps	capable services

The right choice depends on project requirements.

---

47. A Beginner's Learning Path

If you want to become a Python web developer, learn in this order:

```
Python Fundamentals
        ↓
Functions
        ↓
Data Structures
        ↓
OOP
        ↓
Exceptions
        ↓
Files and JSON
        ↓
Git
        ↓
HTTP Basics
        ↓
HTML + CSS + JavaScript Basics
        ↓
Choose Django / Flask / FastAPI
        ↓
Routing
        ↓
Requests and Responses
        ↓
Templates or APIs
        ↓
Databases + SQL
        ↓
Authentication
        ↓
Testing
```

```
        ↓
Security
        ↓
Deployment
        ↓
Monitoring
```

Do not try to memorize an entire framework before understanding HTTP and backend fundamentals.

---

48. Five Minute Challenge

Build a tiny API with one endpoint:

```
GET /hello
```

It should return:

```
{
    "message": "Hello from Python"
}
```

Then add:

```
GET /hello/{name}
```

For:

```
/hello/Hafsa
```

return:

```
{
    "message": "Hello Hafsa"
}
```

Your goal is to understand:

```
Request
→ Route
→ Python Function
→ Response
```

---

49. Mini Project: Resource API

Build a small API for a learning resource platform.

Create:

```
GET    /resources
GET    /resources/{id}
POST   /resources
PATCH  /resources/{id}
DELETE /resources/{id}
```

Each resource could contain:

```
{
    "id": 1,
    "title": "Python Security Basics",
    "category": "Python",
    "type": "PDF"
}
```

Start with an in-memory Python list.

Then upgrade the project:

```
Version 1
In-memory list

Version 2
Database

Version 3
Validation

Version 4
Authentication

Version 5
Tests

Version 6
Deployment
```

This progression teaches the architecture rather than just framework syntax.

---

50. Exercises

Exercise 1

Explain the difference between:

```
Client
Server
Backend
Frontend
```

Exercise 2

Explain what happens when a browser requests:

```
GET /products/10
```

Exercise 3

Create three routes:

```
/
about
resources
```

Exercise 4

Create an API that returns a list of three Python resources.

Exercise 5

Add a dynamic route:

```
/resources/{id}
```

Exercise 6

Add validation to a resource creation endpoint.

Exercise 7

Connect your API to a database.

Exercise 8

Write tests for:

```
Successful request
Invalid request
Missing resource
```

---

51. Mini Quiz

1. Where does Python usually run in a Python web application?

A. Browser

B. Server

C. HTML file

D. CSS file

Answer: B

2. What does GET generally represent?

A. Delete data

B. Read/retrieve data

C. Encrypt data

D. Restart server

Answer: B

3. What does a route connect?

A. Database to CSS

B. URL/request to application code

C. Python to RAM

D. Browser to keyboard

Answer: B

4. What does JSON commonly represent in an API response?

A. Structured data

B. CSS styling

C. Python bytecode

D. Database hardware

Answer: A

5. What is authentication?

A. Determining what a user can do

B. Determining who a user is

C. Compressing a response

D. Creating a database

Answer: B

6. What is authorization?

A. Determining what an authenticated user is allowed to do

B. Determining the user's IP address

C. Formatting JSON

D. Creating HTML

Answer: A

7. What is FastAPI commonly used for?

A. Image editing

B. API and web application development

C. Operating system installation

D. Spreadsheet formatting

Answer: B

---

52. Interview Questions

Beginner

1. What is web development?
2. What is a client and what is a server?
3. What is HTTP?
4. What is a Python web framework?
5. What is routing?
6. What is an API?
7. What is JSON?
8. What are HTTP methods?
9. What is the difference between GET and POST?
10. What is a status code?

Intermediate

11. What is middleware?
12. What is CORS?
13. What is an ORM?
14. What is authentication?
15. What is authorization?
16. What is the difference between WSGI and ASGI?
17. What is synchronous vs asynchronous programming?
18. Why should secrets be stored outside source code?
19. What is the role of a reverse proxy?
20. Why should incoming request data be validated?

Practical

21. How would you design a resource API?
22. How would you handle a missing database record?
23. How would you protect an authenticated endpoint?
24. How would you test a Python API?
25. How would you deploy a Python web application?

---

53. Cheat Sheet

```
Web Development
→ Building software that communicates over the web

Client
→ Usually the browser

Server
→ Machine running backend software

Backend
→ Server-side application logic

HTTP
→ Communication protocol used by web clients and servers

GET
→ Retrieve data

POST
→ Create/send data

PUT
→ Replace/update resource

PATCH
→ Partially update resource

DELETE
→ Delete resource

Route
→ Maps a request path to application code
```

```
Request
→ Information sent by client

Response
→ Information returned by server

JSON
→ Common API data format

Status Code
→ Communicates request result

200
→ Success

201
→ Created

400
→ Bad Request

401
→ Authentication required

403
→ Forbidden

404
→ Not Found

500
→ Server Error

Django
→ Full-featured Python web framework

Flask
→ Lightweight Python web framework

FastAPI
→ Modern API-focused Python framework
```

```
ORM
→ Maps application objects to database records

Middleware
→ Code that processes requests/responses around application handlers

WSGI
→ Interface commonly used by traditional Python web applications

ASGI
→ Interface supporting modern async-capable applications

Authentication
→ Who are you?

Authorization
→ What are you allowed to do?

Deployment
→ Making the application available in a real environment
```

54. Key Takeaways

1. Python usually runs on the server, not inside the browser.
2. Browsers and servers communicate using HTTP.
3. A Python web framework maps requests to application code.
4. Routes determine which code handles a request.
5. Requests can contain paths, query parameters, headers, cookies, and bodies.
6. Responses can contain HTML, JSON, files, and status codes.
7. Python applications commonly communicate with databases and external APIs.
8. Django, Flask, and FastAPI solve different web development needs.
9. Authentication identifies users; authorization controls what they can access.
10. Validation and security are essential because client input cannot be trusted.
11. Testing, logging, configuration, and deployment are part of real web development.
12. Understanding the request → logic → data → response flow is more valuable than memorizing framework syntax.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app