HTTP and APIs with Python

Introduction

Modern applications constantly communicate with other systems.

A website might request:

```
User information
Product data
Weather data
Payment status
AI responses
GitHub repositories
```

A mobile application might communicate with a backend server.

A Python application might communicate with an external service.

The technology that makes much of this communication possible is **HTTP**.

An **API** provides a structured way for applications to communicate with each other.

The basic mental model is:

```
Python Application
        ↓
HTTP Request
        ↓
API
        ↓
HTTP Response
        ↓
Python Application
```

Understanding HTTP and APIs is essential before working with frameworks such as Django, Flask, or FastAPI.

---

1. What Is HTTP?

HTTP stands for:

HyperText Transfer Protocol

It is a protocol that defines how clients and servers communicate over a network.

For example:

```
Python Program
      ↓
HTTP Request
      ↓
example.com
      ↓
HTTP Response
      ↓
Python Program
```

HTTP is not specific to Python.

It is used by:

```
Web browsers
Mobile applications
Backend services
APIs
Cloud services
Command-line tools
```

Python simply provides libraries that make HTTP communication convenient.

---

2. Client and Server

HTTP communication usually involves two sides.

Client

The client sends the request.

Examples:

```
Browser
Mobile app
Python program
JavaScript application
```

Server

The server receives the request and sends a response.

For example:

```
Client
  ↓
GET /users
```

```
  ↓
Server
  ↓
JSON response
```

A Python application can act as either side.

Python can:

```
Send HTTP requests → act as a client
Receive HTTP requests → act as a server
```

---

3. What Is an API?

API stands for:

Application Programming Interface

An API provides a defined way for one piece of software to communicate with another.

For example, imagine a weather service provides:

```
GET /weather?city=Lahore
```

Your Python program can request that endpoint and receive weather data.

Conceptually:

```
Your Python App
       ↓
Weather API
       ↓
JSON
       ↓
Your Python App
```

The API acts as a contract between the two systems.

---

4. Why Do APIs Exist?

Without APIs, applications would have to directly understand the internal implementation of other applications.

An API hides those internal details.

For example, an application might expose:

```
GET /users/42
```

You don't need to know:

```
Which database?
Which database tables?
Which programming language?
Which server?
Which internal functions?
```

You only need to know the API contract.

```
Request
→ Endpoint
→ Parameters
→ Authentication
→ Response format
```

---

5. HTTP Request

An HTTP request can contain several pieces of information:

```
Method
URL
Headers
Query parameters
Path parameters
Body
```

A simplified request looks like:

```
GET /users/42 HTTP/1.1
Host: api.example.com
Accept: application/json
```

The major components are:

```
GET
→ HTTP method

/users/42
→ path

api.example.com
→ host

Accept: application/json
→ header
```

---

6. HTTP Response

The server sends a response.

For example:

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

followed by:

```
{
    "id": 42,
    "name": "Hafsa"
}
```

The response contains:

```
Status code
Headers
Response body
```

---

7. HTTP Methods

HTTP provides methods that describe what the client wants to do.

GET

Retrieve information.

```
GET /users
```

Meaning:

> Give me the users.

POST

Create or submit data.

```
POST /users
```

PUT

Replace a resource.

```
PUT /users/42
```

PATCH

Partially update a resource.

```
PATCH /users/42
```

DELETE

Delete a resource.

```
DELETE /users/42
```

A useful mental model:

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

This is a common REST-style convention, although APIs do not have to follow it perfectly.

---

8. URLs and Endpoints

An API exposes specific URLs called **endpoints**.

For example:

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

might be an endpoint for users.

Another:

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

might represent products.

An endpoint can be thought of as:

```
Specific URL
+
Specific HTTP method
```

For example:

```
GET /products
```

and:

```
POST /products
```

can represent different operations even though they use the same path.

---

9. Path Parameters

A path parameter identifies a specific resource.

Example:

```
GET /users/42
```

Here:

```
42
```

is the user ID.

Another example:

```
GET /products/100
```

means:

> Retrieve product 100.

Path parameters are useful when the resource identity is part of the URL.

---

10. Query Parameters

Query parameters provide additional options.

Example:

```
GET /products?category=books
```

Another:

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

Query parameters are commonly used for:

```
Filtering
Searching
Pagination
Sorting
Optional settings
```

For example:

```
/products?search=python
```

could mean:

> Find products matching "python".

---

11. Request Headers

Headers provide additional information about the request.

Example:

```
Accept: application/json
```

means the client prefers JSON.

Another common header:

```
Content-Type: application/json
```

tells the server that the request body contains JSON.

Headers can also contain authentication information:

```
Authorization: Bearer <token>
```

Common headers include:

```
Accept
Content-Type
Authorization
User-Agent
Cache-Control
```

---

12. Request Body

Some requests send data in the body.

For example:

```
POST /users
Content-Type: application/json
```

Body:

```
{
    "name": "Hafsa",
    "email": "user@example.com"
}
```

The server reads this data and processes it.

GET requests usually do not use a request body for normal API operations. Query parameters or path parameters are generally used for retrieval options.

---

13. JSON and APIs

JSON stands for:

JavaScript Object Notation

It is one of the most common formats used by modern APIs.

Example:

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

JSON supports:

```
Objects
Arrays
Strings
Numbers
Booleans
null
```

Example array:

```
[
    {
        "id": 1,
        "name": "Python"
    },
    {
        "id": 2,
        "name": "JavaScript"
    }
]
```

Python can easily convert JSON into Python data structures.

---

14. JSON in Python

Python includes the built-in `json` module.

```
import json

data = '{"name": "Hafsa", "age": 25}'

user = json.loads(data)

print(user["name"])
```

Output:

```
Hafsa
```

`json.loads()` converts JSON text into Python data.

To convert Python data into JSON:

```
import json

user = {
    "name": "Hafsa",
    "age": 25
}

data = json.dumps(user)

print(data)
```

The distinction is:

```
json.loads()
JSON → Python

json.dumps()
Python → JSON
```

---

15. Python as an HTTP Client

Python applications often need to call external APIs.

For example:

```
Python application
       ↓
GitHub API
       ↓
JSON
```

```
        ↓
Python application
```

Popular HTTP libraries include:

```
requests
httpx
urllib
```

For beginner-friendly synchronous HTTP requests, `requests` is commonly used.

---

16. Installing Requests

Install the package with:

```
pip install requests
```

Then:

```
import requests
```

Remember that `requests` is a third-party package, unlike the built-in `json` module.

---

17. Making a GET Request

A simple request:

```
import requests

response = requests.get(
    "https://api.example.com/users"
)

print(response.status_code)
print(response.text)
```

The result is stored in `response`.

Important properties include:

```
response.status_code
response.text
response.headers
```

---

18. Reading JSON Responses

If the API returns JSON, `requests` provides:

```
response.json()
```

Example:

```
import requests

response = requests.get(
    "https://api.example.com/users"
)

data = response.json()

print(data)
```

If the API returns:

```
{
    "name": "Hafsa"
}
```

Python receives a dictionary-like structure:

```
{
    "name": "Hafsa"
}
```

Then:

```
print(data["name"])
```

produces:

```
Hafsa
```

---

19. Always Check the Response

Do not blindly assume that a request succeeded.

A basic approach:

```
import requests

response = requests.get(
    "https://api.example.com/users"
)
```

```
if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print("Request failed:", response.status_code)
```

This prevents your application from treating an error response as valid data.

---

20. `raise_for_status()`

`requests` provides a convenient method:

```
response.raise_for_status()
```

Example:

```
import requests

response = requests.get(
    "https://api.example.com/users"
)

response.raise_for_status()

data = response.json()

print(data)
```

For unsuccessful HTTP status codes, this raises an exception.

This is often cleaner than manually checking every status code.

---

21. Handling Request Errors

Network communication can fail.

Possible problems include:

```
No internet connection
DNS failure
Timeout
Server unavailable
Invalid URL
Connection failure
```

Use exception handling:

```
import requests

try:
    response = requests.get(
        "https://api.example.com/users",
        timeout=5
    )

    response.raise_for_status()

    data = response.json()

except requests.RequestException as error:
    print("Request failed:", error)
```

The important idea is:

```
Network operation
      ↓
Could fail
      ↓
Handle failure
```

---

22. Why Timeouts Matter

Never assume a network request will finish immediately.

Bad:

```
requests.get(url)
```

Better:

```
requests.get(url, timeout=5)
```

A timeout prevents your application from waiting indefinitely when a remote service becomes unavailable or unresponsive.

---

23. Sending Query Parameters

Instead of manually constructing URLs:

```
import requests

params = {
    "search": "python",
    "page": 2
}

response = requests.get(
    "https://api.example.com/resources",
    params=params
)
```

The library constructs the query string appropriately.

Conceptually:

```
/resources?search=python&page=2
```

Using `params` is safer and cleaner than manually concatenating user-provided values into URLs.

---

24. Sending JSON Data

For a POST request:

```
import requests

user = {
    "name": "Hafsa",
    "email": "user@example.com"
}

response = requests.post(
    "https://api.example.com/users",
    json=user
)

response.raise_for_status()

print(response.json())
```

The `json=` argument automatically handles JSON encoding and the appropriate content type.

---

25. Sending Form Data

Some APIs expect form data instead of JSON.

Example:

```
import requests

data = {
    "username": "hafsa",
    "password": "example"
}

response = requests.post(
    "https://example.com/login",
    data=data
)
```

The format expected by the API should determine whether you use:

```
json=
```

or:

```
data=
```

---

26. Sending Headers

You can provide custom headers:

```
import requests

headers = {
    "Accept": "application/json"
}

response = requests.get(
    "https://api.example.com/users",
    headers=headers
)
```

For authenticated APIs:

```
headers = {
    "Authorization": "Bearer YOUR_TOKEN"
}
```

Never hard-code real production secrets into source code.

---

27. API Authentication

APIs often require authentication.

Common approaches include:

```
API keys
Bearer tokens
OAuth
Session cookies
JWT
```

For example:

```
headers = {
    "Authorization": f"Bearer {token}"
}
```

The exact authentication method depends on the API.

Always read the API's authentication documentation instead of assuming every API uses the same method.

---

28. API Keys

An API might provide a key such as:

```
abc123...
```

The client sends it according to the API's requirements.

For example:

```
headers = {
    "X-API-Key": api_key
}
```

or sometimes:

```
params = {
    "api_key": api_key
}
```

Do not assume where an API key belongs.

Follow the API documentation.

---

29. Environment Variables for API Keys

Never write production secrets directly in code.

Avoid:

```
API_KEY = "real-secret-key"
```

Instead:

```
import os

api_key = os.getenv("API_KEY")
```

Then:

```
headers = {
    "Authorization": f"Bearer {api_key}"
}
```

Your environment can provide:

```
API_KEY=...
```

This keeps secrets separate from source code.

---

30. HTTP Status Codes

Status codes tell the client what happened.

Successful responses

```
200 OK
201 Created
204 No Content
```

Client errors

```
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
409 Conflict
429 Too Many Requests
```

Server errors

```
500 Internal Server Error
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
```

A useful mental model:

```
2xx → Success
3xx → Redirection
4xx → Client/request problem
5xx → Server-side problem
```

Note that `401 Unauthorized` is conventionally used when authentication is missing or invalid, while `403 Forbidden` generally means the server understood the request but refuses to authorize it.

---

31. REST APIs

REST stands for:

Representational State Transfer

REST is an architectural style for designing networked applications.

A REST-style API often models resources.

For example:

```
/users
/products
/orders
/resources
```

Operations can then use HTTP methods:

```
GET    /resources
POST   /resources
GET    /resources/10
PATCH  /resources/10
DELETE /resources/10
```

REST is a design approach, not a Python library.

---

32. Resource-Oriented Thinking

Instead of thinking:

```
/getAllUsers
/createUser
/deleteUser
```

a REST-style design often thinks in terms of resources:

```
/users
```

Then HTTP methods communicate the operation:

```
GET    /users
POST   /users
```

For a specific user:

```
GET    /users/42
PATCH  /users/42
DELETE /users/42
```

This makes APIs easier to reason about.

---

33. API Pagination

An API may have thousands of records.

Returning everything at once can be inefficient.

Instead:

```
GET /resources?page=1
```

or:

```
GET /resources?limit=20&offset=40
```

The exact pagination strategy depends on the API.

A response might contain:

```
{
    "items": [...],
    "page": 2,
    "total": 100
}
```

Pagination helps control:

```
Response size
Database load
```

```
Network usage
Client processing
```

---

34. API Filtering

APIs commonly allow filtering.

Example:

```
GET /resources?category=python
```

Python:

```
params = {
    "category": "python"
}

response = requests.get(
    "https://api.example.com/resources",
    params=params
)
```

The API decides how the filter is interpreted.

---

35. API Rate Limits

External APIs may restrict how many requests you can make.

For example:

```
100 requests per minute
```

If you exceed the limit, the server might return:

```
429 Too Many Requests
```

A good client should handle rate limits rather than continuously retrying immediately.

Some APIs provide a `Retry-After` header that indicates when another attempt may be appropriate.

---

36. Retries

Some failures are temporary.

For example:

```
Temporary network failure
503 Service Unavailable
```

A retry strategy can attempt the request again.

But retries must be controlled.

Avoid:

```
Request fails
 ↓
Retry immediately
 ↓
Fail
 ↓
Retry immediately
 ↓
Repeat forever
```

A better approach can use:

```
Limited attempts
+
Increasing delay
+
Appropriate status checks
```

This is commonly called **exponential backoff**.

Not every request should automatically be retried. For example, blindly retrying a POST operation could create duplicate data if the server actually processed the first request but the response was lost.

---

37. Idempotency

An operation is idempotent when repeating the same operation produces the same intended final state.

For example:

```
PUT /users/42
```

setting the same user's name to `"Hafsa"` repeatedly can be idempotent.

POST requests often represent creation and may not be idempotent.

Some APIs use an **idempotency key** for operations such as payments so that a client can safely retry without accidentally creating multiple transactions.

This is an important concept when building reliable API integrations.

---

38. HTTP Sessions

If your Python program makes many requests to the same service, `requests.Session()` can maintain certain state and reuse connections.

Example:

```
import requests

with requests.Session() as session:
    session.headers.update({
        "Accept": "application/json"
    })

    response = session.get(
        "https://api.example.com/users"
    )

    response.raise_for_status()

    print(response.json())
```

Sessions can be useful for:

```
Connection reuse
Shared headers
Cookies
Authentication state
Multiple related requests
```

---

39. Cookies

HTTP cookies allow servers and clients to maintain state across requests.

Python can work with cookies through a session:

```
import requests

with requests.Session() as session:
    response = session.get(
```

```
        "https://example.com"
    )

    print(session.cookies)
```

Cookies are commonly used for:

```
Sessions
Authentication
Preferences
Tracking
```

Cookie behavior depends on the application and security configuration.

---

40. `requests` vs `httpx`

Both are useful Python HTTP libraries.

Requests

Simple and mature.

```
import requests

response = requests.get(url)
```

Good for many straightforward synchronous applications.

HTTPX

Supports both synchronous and asynchronous HTTP clients.

Synchronous:

```
import httpx

response = httpx.get(url)
```

Asynchronous:

```
import httpx

async def get_data():
    async with httpx.AsyncClient() as client:
        response = await client.get(url)

        return response.json()
```

Choose the library based on your application's requirements.

---

41. Calling an API with `httpx`

Example:

```
import httpx

response = httpx.get(
    "https://api.example.com/resources",
    timeout=5
)

response.raise_for_status()

data = response.json()

print(data)
```

For async applications:

```
import httpx


async def get_resources():
    async with httpx.AsyncClient() as client:
        response = await client.get(
            "https://api.example.com/resources"
        )

        response.raise_for_status()

        return response.json()
```

Async HTTP clients are particularly useful when your application already uses asynchronous code and needs to make many concurrent I/O requests.

---

42. Building Your Own API with FastAPI

So far, Python has been consuming APIs.

Python can also create APIs.

Example:

```
from fastapi import FastAPI

app = FastAPI()
```

```
@app.get("/resources")
def get_resources():
    return [
        {
            "id": 1,
            "title": "Python Basics"
        },
        {
            "id": 2,
            "title": "HTTP and APIs"
        }
    ]
```

A client can call:

```
GET /resources
```

and receive JSON.

---

43. POST Endpoint with FastAPI

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Resource(BaseModel):
    title: str
    category: str


@app.post("/resources")
def create_resource(resource: Resource):
    return resource
```

A client can send:

```
{
    "title": "HTTP and APIs",
    "category": "Python"
}
```

FastAPI validates the incoming structure using the model.

---

44. API Contract

An API should clearly define:

```
Endpoint
HTTP method
Parameters
Request body
Authentication
Response structure
Status codes
Errors
Rate limits
```

For example:

```
POST /resources

Request:
{
    "title": "HTTP and APIs",
    "category": "Python"
}

Success:
201 Created

Response:
{
    "id": 10,
    "title": "HTTP and APIs",
    "category": "Python"
}
```

This contract allows different applications to communicate without knowing each other's internal implementation.

---

45. API Documentation

Good APIs provide documentation.

It should answer:

```
What endpoint should I call?
Which HTTP method?
Which parameters?
Which headers?
How do I authenticate?
What does the response look like?
What errors can occur?
```

FastAPI can automatically generate interactive API documentation.

This is one reason it is popular for API development.

---

46. API Errors

APIs should provide useful error responses.

For example:

```
{
    "error": "Resource not found"
}
```

or:

```
{
    "error": "Validation failed",
    "fields": {
        "email": "Invalid email address"
    }
}
```

A Python client should not assume every response contains the successful response structure.

This is dangerous:

```
data = response.json()

print(data["user"]["name"])
```

If the API returns an error structure, this could fail.

Check the response first.

---

47. A Reliable API Client

A small reusable client function might look like:

```
import requests


def get_resource(resource_id):
    url = f"https://api.example.com/resources/{resource_id}"

    try:
        response = requests.get(
            url,
            timeout=5
        )

        response.raise_for_status()

        return response.json()

    except requests.RequestException as error:
        print(f"API request failed: {error}")
        return None
```

This provides:

```
Timeout
Error handling
Status checking
JSON parsing
Reusable function
```

For larger applications, API clients should usually be separated into their own modules and use proper logging instead of `print()`.

---

48. API Client Architecture

Instead of calling external APIs throughout your application:

```
requests.get(...)
requests.post(...)
requests.get(...)
```

everywhere, create a dedicated API client.

For example:

```
project/
|
```

```
├── api/
│   └── github_client.py
│
├── services/
│   └── resource_service.py
│
└── main.py
```

Then:

```
Application
    ↓
Service
    ↓
API Client
    ↓
External API
```

This keeps external communication separate from business logic.

---

49. Example: haas.dev Resource API

Imagine haas.dev exposes an API:

```
GET /api/resources
```

A Python client could request:

```
import requests


response = requests.get(
    "https://dev-roast-app.vercel.app/api/resources",
    timeout=5
)

response.raise_for_status()

resources = response.json()

for resource in resources:
    print(resource["title"])
```

The flow is:

```
Python Program
      ↓
GET /api/resources
      ↓
haas.dev API
      ↓
JSON response
      ↓
Python dictionary/list
      ↓
Application logic
```

The Python program does not need to know how the website internally stores its resources. It only needs to understand the API contract.

---

50. HTTP and Security

HTTP communication can involve sensitive information.

Examples:

```
Passwords
API keys
Access tokens
Personal information
Payment information
```

For production applications, communication should generally use:

```
HTTPS
```

HTTPS encrypts HTTP traffic using TLS.

Instead of:

```
http://example.com
```

you generally want:

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

Do not disable TLS certificate verification just to make a failing request work.

Avoid code such as:

```
requests.get(
    url,
```

```
    verify=False
)
```

unless you have a specific controlled reason and understand the security consequences.

---

51. Never Trust API Responses Blindly

Even external services can return:

```
Missing fields
Unexpected data
Invalid values
Error responses
Changed structures
```

Validate important external data before using it.

For critical applications, define schemas or models for expected responses.

This protects your application from assumptions about external systems.

---

52. Common Mistakes

Mistake 1: Ignoring status codes

Bad:

```
data = requests.get(url).json()
```

Better:

```
response = requests.get(url)
response.raise_for_status()
data = response.json()
```

Mistake 2: No timeout

Bad:

```
requests.get(url)
```

Better:

```
requests.get(url, timeout=5)
```

Mistake 3: Hard-coding API keys

Bad:

```
api_key = "secret"
```

Better:

```
import os

api_key = os.getenv("API_KEY")
```

Mistake 4: Building URLs with string concatenation

Avoid:

```
url = base_url + "?search=" + user_input
```

Prefer:

```
params = {
    "search": user_input
}

requests.get(
    base_url,
    params=params
)
```

Mistake 5: Treating every response as JSON

Some responses may be:

```
HTML
Plain text
Empty
Binary data
```

Know what the API promises before calling:

```
response.json()
```

Mistake 6: Retrying everything

A retry can be dangerous for operations that create or charge something.

Understand idempotency before implementing automatic retries.

Mistake 7: Logging secrets

Avoid:

```
print(api_key)
print(access_token)
print(password)
```

Secrets should never appear in normal logs.

---

53. HTTP Mental Model

When working with HTTP, ask:

```
1. Who is the client?

2. Who is the server?

3. What URL is being requested?

4. Which HTTP method is being used?

5. What headers are required?

6. Is there a request body?

7. Is authentication required?

8. What status codes are possible?

9. What does the response look like?

10. What happens if the request fails?
```

This checklist solves many API integration problems.

---

54. Five Minute Challenge

Use Python and `requests` to call a public API.

Your program should:

1. Send a GET request.
2. Set a timeout.
3. Check the status.
4. Parse JSON.
5. Print one useful field.
6. Handle a request failure.

Basic structure:

```
import requests


try:
    response = requests.get(
        "API_URL",
        timeout=5
    )

    response.raise_for_status()

    data = response.json()

    print(data)

except requests.RequestException as error:
    print("Request failed:", error)
```

Replace `API_URL` with an API you are allowed to use.

---

55. Mini Project: Python API Client

Build a command-line application that consumes an API.

For example:

```
Python Resource Explorer
```

Features:

```
1. Search resources
2. View resource details
3. Filter by category
4. Handle API errors
5. Display results
```

Suggested architecture:

```
CLI
 ↓
Service
 ↓
API Client
```

```
 ↓
HTTP API
```

Start simple.

Then add:

```
Timeouts
Authentication
Pagination
Caching
Retries
Logging
Tests
```

This project teaches how real applications consume external APIs.

---

56. Exercises

Exercise 1

Explain the difference between:

```
HTTP
API
Endpoint
URL
```

Exercise 2

Write a Python GET request using `requests`.

Exercise 3

Print:

```
Status code
Response headers
Response JSON
```

Exercise 4

Send query parameters using:

```
params={}
```

Exercise 5

Send a JSON body using:

```
json={}
```

Exercise 6

Add:

```
timeout=5
```

Exercise 7

Handle:

```
Connection error
HTTP error
Invalid JSON
```

Exercise 8

Create a FastAPI endpoint:

```
GET /resources
```

Exercise 9

Create:

```
POST /resources
```

with validation.

Exercise 10

Build a small Python client that communicates with your API.

---

57. Mini Quiz

1. What does HTTP provide?

A. A database

B. A protocol for web communication

C. A Python framework

D. A programming language

Answer: B

2. What is an API?

A. A structured interface for software communication

B. A database table

C. A browser

D. A Python variable

Answer: A

3. Which method is commonly used to retrieve data?

A. POST

B. GET

C. DELETE

D. PATCH

Answer: B

4. What does `response.json()` do?

A. Sends a request

B. Converts a JSON response into Python data

C. Starts a server

D. Encrypts a response

Answer: B

5. Why should HTTP requests have timeouts?

A. To make JSON smaller

B. To prevent indefinite waiting

C. To create authentication

D. To change the HTTP method

Answer: B

6. What does status code 404 generally mean?

A. Success

B. Created

C. Not Found

D. Server started

Answer: C

7. What does status code 429 generally indicate?

A. Unauthorized

B. Not Found

C. Too Many Requests

D. Created

Answer: C

8. Where should production API secrets generally be stored?

A. Public Git repository

B. Source code comments

C. Environment/secret configuration

D. HTML

Answer: C

---

58. Interview Questions

Beginner

1. What is HTTP?
2. What is an API?
3. What is an endpoint?
4. What is the difference between a client and a server?
5. What is JSON?
6. What are HTTP methods?
7. What is the difference between GET and POST?
8. What is a status code?
9. What are query parameters?
10. What are request headers?

Intermediate

11. What is REST?
12. What is the difference between PUT and PATCH?
13. What is authentication?
14. What is authorization?
15. What is CORS?
16. Why are timeouts important?
17. What is API rate limiting?
18. What is exponential backoff?
19. What is idempotency?
20. Why should API keys not be hard-coded?

Practical

21. How would you consume an external API in Python?
22. How would you handle an API timeout?

23. How would you handle a 429 response?
24. How would you design a reusable API client?
25. How would you test an API integration?
26. How would you prevent secrets from appearing in logs?
27. When would you choose `requests` over `httpx`?
28. When would asynchronous HTTP requests be useful?

---

59. Cheat Sheet

```
HTTP
→ Protocol for communication between clients and servers

API
→ Interface that allows software systems to communicate

Endpoint
→ Specific API operation identified by a method + URL/path

GET
→ Retrieve data

POST
→ Create/submit data

PUT
→ Replace a resource

PATCH
→ Partially update a resource

DELETE
→ Delete a resource

Headers
→ Additional request/response metadata

Query Parameters
→ Optional values in the URL

Path Parameters
→ Values identifying a resource in the URL path
```

Request Body
→ Data sent with a request

JSON
→ Common structured API data format

200
→ OK

201
→ Created

204
→ No Content

400
→ Bad Request

401
→ Authentication required

403
→ Forbidden

404
→ Not Found

409
→ Conflict

429
→ Too Many Requests

500
→ Internal Server Error

requests
→ Popular synchronous Python HTTP library

httpx
→ HTTP client supporting sync and async usage

```
response.json()
→ Parse JSON response

raise_for_status()
→ Raise exception for unsuccessful HTTP status

timeout
→ Maximum waiting period for a request

Session
→ Reusable HTTP client state/connections

REST
→ Architectural style for network APIs

HTTPS
→ HTTP secured with TLS

Authentication
→ Verify identity

Authorization
→ Determine permissions

Idempotency
→ Repeating an operation produces the same intended final state
```

60. Key Takeaways

1. HTTP is the communication protocol behind much of the web.
2. An API provides a structured interface between software systems.
3. Python can consume APIs and build APIs.
4. HTTP requests contain methods, URLs, headers, and sometimes bodies.
5. HTTP responses contain status codes, headers, and response data.
6. JSON is one of the most common formats used by APIs.
7. `requests` provides a simple way to make HTTP requests from Python.
8. `httpx` is useful when synchronous and asynchronous HTTP clients are needed.
9. Always consider timeouts, errors, authentication, rate limits, and retries.
10. Never hard-code production secrets into source code.
11. API clients should validate responses instead of assuming they are always successful.

12. REST commonly models APIs around resources and HTTP methods.
13. Good API integrations treat network communication as something that can fail.
14. Understanding HTTP makes Python web frameworks much easier to understand.

Visit haas.dev for more resources and guides.

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