Making API Requests with Python

Introduction

APIs allow Python programs to communicate with external services.

For example, a Python application might:

```
Fetch GitHub repositories
Send data to a payment service
Get weather information
Call an AI API
Upload a file
Create a database record through an API
Retrieve user information
```

The previous topic explained **HTTP and APIs** conceptually.

This guide focuses on the practical skill:

> **How do you actually make API requests from Python?**

The most important library for beginners is `requests`.

The basic flow is:

```
Python Code
    ↓
Build Request
    ↓
Send HTTP Request
    ↓
Receive Response
    ↓
Check Status
    ↓
Read Response Data
    ↓
Use Data in Your Application
```

---

1. What Does "Making an API Request" Mean?

Suppose an API provides:

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

Your Python program can ask:

> Give me the users.

Python sends an HTTP request:

```
Python
  ↓
GET /users
  ↓
API Server
```

The server might respond:

```
[
    {
        "id": 1,
        "name": "Hafsa"
    },
    {
        "id": 2,
        "name": "Asim"
    }
]
```

Python receives this response and can work with it.

---

2. The `requests` Library

Python's standard library contains networking tools, but `requests` provides a much simpler interface for common HTTP requests.

Install it:

```
pip install requests
```

Then import it:

```
import requests
```

A simple GET request:

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

The returned object is a `Response`.

---

3. Understanding the Response Object

After making a request:

```
response = requests.get(url)
```

you can inspect:

```
response.status_code
response.headers
response.text
response.content
response.json()
```

For example:

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

Each property provides different information.

---

4. Checking the Status Code

Never assume that a request succeeded.

```
import requests

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

print(response.status_code)
```

A successful request might return:

```
200
```

An unavailable resource might return:

```
404
```

An authentication problem might return:

```
401
```

A server failure might return:

```
500
```

A useful pattern is:

```
if response.status_code == 200:
    print("Success")
else:
    print("Request failed")
```

---

5. Using `raise_for_status()`

For most application code, this is cleaner:

```
response.raise_for_status()
```

Example:

```
import requests

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

response.raise_for_status()

print(response.json())
```

For unsuccessful HTTP responses, `raise_for_status()` raises an exception.

This lets you handle errors with normal Python exception handling.

---

6. Reading JSON

If the API returns JSON:

```
data = response.json()
```

For example, if the API returns:

```
{
    "name": "Hafsa",
    "role": "Developer"
}
```

then:

```
data = response.json()

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

outputs:

```
Hafsa
```

The JSON object becomes a Python dictionary.

---

7. JSON Arrays

An API can also return a list.

Example:

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

Python:

```
data = response.json()

for item in data:
    print(item["name"])
```

Output:

```
Python
JavaScript
```

Remember:

```
JSON object → Python dictionary
JSON array  → Python list
```

---

8. Reading Plain Text

Not every API returns JSON.

You can access the raw response as text:

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

For example:

```
Hello from the server
```

`response.text` is useful when the response is:

```
Plain text
HTML
Other text-based content
```

Do not automatically call:

```
response.json()
```

unless the response is expected to contain JSON.

---

9. Reading Binary Data

For files, images, PDFs, or other binary content:

```
content = response.content
```

For example:

```
with open("file.pdf", "wb") as file:
    file.write(response.content)
```

The important distinction is:

```
response.text
→ Text

response.json()
→ Parsed JSON

response.content
→ Raw bytes
```

---

10. GET Requests

GET requests retrieve data.

```
import requests

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

response.raise_for_status()

products = response.json()

print(products)
```

Conceptually:

```
GET
 ↓
Retrieve information
 ↓
Response
```

---

11. Query Parameters

Suppose an API supports:

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

Instead of manually constructing the URL, use `params`.

```
import requests

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

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

`requests` constructs the query string for you.

You can inspect the final URL:

```
print(response.url)
```

This might produce:

```
https://api.example.com/products?category=python&page=2
```

---

12. Why `params` Is Better Than String Concatenation

Avoid:

```
url = (
    "https://api.example.com/products"
    + "?search="
    + search_term
)
```

This becomes difficult to maintain and can create URL-encoding problems.

Prefer:

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

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

Let the HTTP library handle URL encoding.

---

13. Multiple Query Parameters

You can send multiple parameters:

```
params = {
    "search": "python",
    "category": "programming",
    "page": 2,
    "limit": 20
}

response = requests.get(
    url,
    params=params
)
```

The resulting URL will contain all of them.

This is useful for:

```
Search
Filtering
Pagination
Sorting
Limits
Optional settings
```

---

14. Path Parameters

Path parameters are usually part of the URL itself.

Suppose you want user `42`:

```
user_id = 42

url = f"https://api.example.com/users/{user_id}"

response = requests.get(url)
```

The final URL becomes:

```
https://api.example.com/users/42
```

Use:

```
Path parameter
→ Resource identity

Query parameter
→ Filtering/options
```

For example:

```
/users/42
```

identifies a user.

While:

```
/users?role=developer
```

filters users.

---

15. Request Headers

Headers provide additional information.

Example:

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

response = requests.get(
    url,
    headers=headers
)
```

A common header is:

```
Content-Type
```

which tells the server what format the request body uses.

Another is:

```
Authorization
```

which can contain authentication information.

---

16. Sending JSON with POST

Suppose the API expects:

```
{
    "title": "Python Security",
    "category": "Python"
}
```

Use the `json=` argument:

```
import requests

data = {
    "title": "Python Security",
    "category": "Python"
}

response = requests.post(
    "https://api.example.com/resources",
```

```
    json=data
)

response.raise_for_status()

print(response.json())
```

This is preferable to manually converting the dictionary with `json.dumps()` for a normal JSON API request.

---

17. `json=` vs `data=`

These are different.

JSON request

```
requests.post(
    url,
    json=data
)
```

Form data

```
requests.post(
    url,
    data=data
)
```

Use whichever format the API expects.

For a JSON API, generally use:

```
json=data
```

---

18. Sending Form Data

Some APIs expect form-encoded data.

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

response = requests.post(
    url,
```

```
    data=data
)
```

The API documentation should tell you which format is required.

---

19. Sending Custom Headers and JSON Together

You can combine options:

```
headers = {
    "Authorization": f"Bearer {token}",
    "Accept": "application/json"
}

data = {
    "title": "Python APIs",
    "category": "Python"
}

response = requests.post(
    url,
    headers=headers,
    json=data,
    timeout=5
)
```

This is a common pattern for authenticated APIs.

---

20. Authentication with Bearer Tokens

Many APIs use bearer tokens.

The request may look like:

```
Authorization: Bearer TOKEN
```

Python:

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

response = requests.get(
    url,
```

```
    headers=headers
)
```

The exact authentication mechanism depends on the API.

Always follow the API's documentation.

---

21. API Keys

Some APIs use API keys.

For example:

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

Others may expect:

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

Do not assume where the key belongs.

The API documentation defines this.

---

22. Never Hard-Code Secrets

Avoid:

```
api_key = "real-production-key"
```

Instead:

```
import os

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

Then:

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

This keeps secrets outside the source code.

Also make sure files containing secrets are not committed to Git.

---

23. `.env` Files During Development

A local project might use:

```
.env
```

with:

```
API_KEY=your-secret
```

A library such as `python-dotenv` can load it.

Install:

```
pip install python-dotenv
```

Then:

```
import os

from dotenv import load_dotenv

load_dotenv()

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

Do not commit `.env` when it contains real secrets.

Add it to `.gitignore`:

```
.env
```

---

24. Timeouts

Network requests can take longer than expected.

Always consider specifying a timeout:

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

This means the request should not wait indefinitely.

Without timeouts, an unavailable service can cause your application to remain stuck waiting.

---

25. Connecting Timeouts and Read Timeouts

`requests` also allows more specific timeout control:

```
response = requests.get(
    url,
    timeout=(3, 10)
)
```

This represents:

```
3 seconds → connection timeout
10 seconds → read timeout
```

This can provide more control than a single timeout value.

---

26. Handling Network Errors

Network operations can fail for many reasons.

Use exception handling:

```
import requests

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

    response.raise_for_status()

except requests.RequestException as error:
    print(f"Request failed: {error}")
```

`RequestException` is the general base exception for many `requests` errors.

---

27. Handling HTTP Errors Separately

You can distinguish HTTP failures:

```
import requests

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

```
    )

    response.raise_for_status()

except requests.HTTPError as error:
    print("HTTP error:", error)

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

This can be useful when your application needs different behavior for server responses versus connection problems.

---

28. Handling JSON Errors

Even when an API usually returns JSON, a failure response might contain HTML or another format.

Instead of blindly doing:

```
data = response.json()
```

you can handle parsing errors.

```
try:
    data = response.json()
except ValueError:
    print("Response was not valid JSON")
```

In real applications, define the expected API contract and validate important responses.

---

29. A Robust Basic GET Request

A practical pattern:

```
import requests


def get_resources():
    try:
        response = requests.get(
            "https://api.example.com/resources",
            timeout=5
        )
```

```
        response.raise_for_status()

        return response.json()

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

The flow is:

```
Call API
   ↓
Timeout protection
   ↓
Check HTTP status
   ↓
Parse JSON
   ↓
Return data
```

---

30. PUT Requests

PUT is commonly used to replace a resource.

```
data = {
    "title": "Updated Python Guide",
    "category": "Python"
}

response = requests.put(
    "https://api.example.com/resources/10",
    json=data,
    timeout=5
)

response.raise_for_status()
```

The exact semantics depend on the API.

---

31. PATCH Requests

PATCH is commonly used for partial updates.

```
data = {
    "title": "Updated Title"
}

response = requests.patch(
    "https://api.example.com/resources/10",
    json=data,
    timeout=5
)

response.raise_for_status()
```

Only the provided field may be changed.

Again, the API's contract determines the exact behavior.

---

32. DELETE Requests

To delete a resource:

```
response = requests.delete(
    "https://api.example.com/resources/10",
    timeout=5
)

response.raise_for_status()
```

A successful DELETE might return:

```
204 No Content
```

In that case, there may be no JSON body to parse.

Do not automatically call:

```
response.json()
```

after every request.

---

33. Checking for an Empty Response

For example:

```
response.raise_for_status()
```

```
if response.status_code != 204:
    data = response.json()
```

This prevents attempting to parse an empty response as JSON.

---

34. Working with Response Headers

Response headers can contain useful information:

```
response = requests.get(url)

print(response.headers)
```

You can access a specific header:

```
content_type = response.headers.get(
    "Content-Type"
)

print(content_type)
```

Headers may provide information about:

```
Content type
Caching
Rate limits
Server behavior
Retry timing
Pagination
```

---

35. Rate Limit Information

Some APIs return headers such as:

```
X-RateLimit-Remaining
Retry-After
```

For example:

```
retry_after = response.headers.get(
    "Retry-After"
)
```

The exact headers vary between APIs.

If the server responds with:

```
429 Too Many Requests
```

respect the API's rate-limit rules.

---

36. Retrying Requests

Some temporary failures can be retried.

But retries should be:

```
Limited
Controlled
Delayed
Appropriate for the operation
```

For example, conceptually:

```
Attempt 1
   ↓
Failure
   ↓
Wait
   ↓
Attempt 2
   ↓
Failure
   ↓
Wait longer
   ↓
Attempt 3
```

This is called **exponential backoff** when the delay increases progressively.

Do not blindly retry every request.

A POST that creates something may need idempotency protection before automatic retries are safe.

---

37. Idempotency and Retries

Consider:

```
POST /payments
```

Your application sends the request.

The server processes the payment.

But the network connection fails before your application receives the response.

Your application cannot automatically know whether the payment succeeded.

If it sends the POST again, it could create a duplicate operation.

For operations like payments, APIs may provide an **idempotency key**:

```
headers = {
    "Idempotency-Key": unique_key
}
```

The server can use that key to recognize repeated attempts.

The exact mechanism depends on the API.

---

38. Using a Session

If your application makes multiple requests to the same service, use a session:

```
import requests

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

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

    response.raise_for_status()

    print(response.json())
```

A session can maintain:

```
Cookies
Headers
Connection reuse
Authentication-related state
```

---

39. Creating a Reusable API Client

Instead of scattering requests throughout your application:

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

create a dedicated client.

Example:

```
import requests


class ResourceClient:
    def __init__(self, base_url):
        self.base_url = base_url

    def get_resource(self, resource_id):
        response = requests.get(
            f"{self.base_url}/resources/{resource_id}",
            timeout=5
        )

        response.raise_for_status()

        return response.json()
```

Use it:

```
client = ResourceClient(
    "https://api.example.com"
)

resource = client.get_resource(10)

print(resource)
```

This gives your application a cleaner interface.

---

40. Better API Client Design

A production API client can centralize:

```
Base URL
Authentication
Headers
Timeouts
```

```
Error handling
Retries
Serialization
Logging
```

For example:

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

This keeps network-specific details away from your business logic.

---

41. Separate API Logic from Business Logic

Avoid putting everything into one function:

```
def create_resource():
    # Build URL
    # Authenticate
    # Make HTTP request
    # Parse JSON
    # Validate response
    # Calculate business rules
    # Save database data
    # Send email
```

This becomes difficult to test and maintain.

Instead:

```
Route / CLI
    ↓
Service
    ↓
API Client
    ↓
External API
```

The service decides **what the application should do**.

The API client handles **how it communicates with the external system**.

42. Example: haas.dev Resource Client

Imagine an API for haas.dev resources.

A client could look like:

```
import requests


class HaasResourceClient:
    def __init__(self, base_url):
        self.base_url = base_url.rstrip("/")

    def list_resources(self):
        response = requests.get(
            f"{self.base_url}/resources",
            timeout=5
        )

        response.raise_for_status()

        return response.json()

    def get_resource(self, resource_id):
        response = requests.get(
            f"{self.base_url}/resources/{resource_id}",
            timeout=5
        )

        response.raise_for_status()

        return response.json()
```

Then:

```
client = HaasResourceClient(
    "https://example.com/api"
)

resources = client.list_resources()

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

The rest of the application does not need to know how URLs are constructed or how `requests` works.

---

43. Authentication in a Reusable Client

Instead of repeating authentication headers:

```
class APIClient:
    def __init__(self, base_url, token):
        self.session = requests.Session()

        self.session.headers.update({
            "Authorization": f"Bearer {token}",
            "Accept": "application/json"
        })

        self.base_url = base_url.rstrip("/")
```

Now every request through that session automatically receives the configured headers.

---

44. Handling Authentication Failure

Suppose the server returns:

```
401 Unauthorized
```

Your application should not treat this as a successful API response.

For example:

```
response = session.get(url)

if response.status_code == 401:
    print("Authentication failed")
else:
    response.raise_for_status()
```

For a larger application, authentication failures might trigger:

```
Token refresh
Re-authentication
Logout
User notification
```

depending on the authentication system.

---

45. Calling APIs from a CLI

API requests are not limited to web applications.

You can build:

```
Python CLI
    ↓
API
    ↓
JSON
    ↓
Terminal
```

Example:

```
import requests

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

response.raise_for_status()

for resource in response.json():
    print(resource["title"])
```

This is a simple way to practice API integration.

---

46. Calling APIs from a Web Backend

A Python backend can also call another API.

For example:

```
Browser
   ↓
Your Python Backend
   ↓
External API
   ↓
Your Python Backend
   ↓
Browser
```

This pattern is useful when:

```
API credentials must remain private
External data needs processing
Multiple APIs need to be combined
Business rules must be applied
```

The browser should not receive private API credentials that belong on the server.

---

47. Calling APIs from Background Jobs

An API request can also happen in a background task.

For example:

```
New order
   ↓
Background job
   ↓
Shipping API
   ↓
Shipping label
   ↓
Store result
```

This prevents a long external API call from unnecessarily blocking a user's request.

---

48. Testing API Requests

External APIs make tests more difficult because they depend on the network.

You generally do not want every unit test to make a real external request.

Instead, mock or substitute the external HTTP layer.

Conceptually:

```
Application
    ↓
Mock API
    ↓
Fake response
```

Then you can test:

```
Successful response
404 response
```

```
401 response
500 response
Timeout
Invalid JSON
```

without depending on the real service.

---

49. Logging API Requests Safely

Logging can help diagnose failures:

```
import logging

logger = logging.getLogger(__name__)

logger.info(
    "Requesting resource %s",
    resource_id
)
```

Do not log:

```
Passwords
API keys
Access tokens
Authorization headers
Sensitive personal data
```

Good logs explain what happened without exposing secrets.

---

50. API Request Checklist

Before sending an API request, identify:

```
1. URL
2. HTTP method
3. Path parameters
4. Query parameters
5. Headers
6. Authentication
7. Request body
8. Timeout
9. Expected status codes
10. Response format
```

```
11. Possible errors
12. Rate limits
```

This turns API documentation into an implementation plan.

---

51. Reading API Documentation

When using an unfamiliar API, do not immediately start coding.

First find:

```
Authentication
Base URL
Endpoint
Method
Parameters
Request example
Response example
Error responses
Rate limits
Pagination
Timeout/retry guidance
```

For example, if the documentation says:

```
POST /resources

Headers:
Authorization: Bearer TOKEN

Body:
{
    "title": string,
    "category": string
}

Success:
201 Created
```

you can translate that into:

```
response = requests.post(
    f"{base_url}/resources",
    headers={
        "Authorization": f"Bearer {token}"
```

```
    },
    json={
        "title": title,
        "category": category
    },
    timeout=5
)
```

The documentation is the source of truth.

---

52. Common Mistakes

Mistake 1: No timeout

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

Prefer:

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

Mistake 2: Ignoring status codes

```
data = response.json()
```

without checking whether the request succeeded.

Prefer:

```
response.raise_for_status()
data = response.json()
```

when the API contract supports this pattern.

Mistake 3: Hard-coding secrets

```
token = "real-token"
```

Use environment or secret configuration instead.

Mistake 4: Parsing every response as JSON

A response may be:

```
204 No Content
HTML
```

```
Plain text
Binary data
```

Know the expected response format.

Mistake 5: Building query strings manually

Use:

```
params={}
```

instead of manually concatenating values.

Mistake 6: Retrying unsafe operations

Understand idempotency before automatically retrying requests that create or trigger actions.

Mistake 7: Mixing API and business logic

Separate external communication from application rules.

Mistake 8: Exposing API credentials in frontend code

Private credentials should generally remain on a trusted backend.

---

53. Five Minute Challenge

Create a function:

```
get_resource(resource_id)
```

It should:

1. Build the URL using `resource_id`.
2. Send a GET request.
3. Use a timeout.
4. Check the HTTP status.
5. Parse JSON.
6. Return the result.
7. Handle request failures.

Target structure:

```
import requests
```

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

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

        response.raise_for_status()

        return response.json()

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

---

54. Mini Project: Python API Explorer

Build a command-line application called:

```
API Explorer
```

It should allow the user to:

```
1. List resources
2. View a resource
3. Search resources
4. Filter resources
5. Create a resource
6. Update a resource
7. Delete a resource
```

Suggested architecture:

```
CLI
 ↓
Service
 ↓
API Client
```

```
  ↓
HTTP API
```

Start with:

```
GET
```

Then add:

```
POST
PATCH
DELETE
```

Finally add:

```
Authentication
Timeouts
Error handling
Logging
Tests
```

---

55. Exercises

Exercise 1

Make a GET request and print:

```
Status code
URL
Response headers
Response body
```

Exercise 2

Send three query parameters using `params`.

Exercise 3

Send a JSON object using `json=`.

Exercise 4

Add an authorization header.

Exercise 5

Add a five-second timeout.

Exercise 6

Handle `RequestException`.

Exercise 7

Handle a `404` response separately.

Exercise 8

Handle an empty `204` response.

Exercise 9

Create a reusable API client class.

Exercise 10

Write tests for:

```
Successful request
404 response
401 response
500 response
Timeout
Invalid JSON
```

---

56. Mini Quiz

1. Which library is commonly used for simple HTTP requests in Python?

A. requests

B. pathlib

C. tkinter

D. random

Answer: A

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

A. Sends JSON

B. Parses a JSON response

C. Starts a server

D. Encrypts JSON

Answer: B

3. Which argument is commonly used to send JSON?

A. `params=`

B. `headers=`

C. `json=`

D. `path=`

Answer: C

4. Which argument is used for query parameters?

A. `params=`

B. `json=`

C. `body=`

D. `query=`

Answer: A

5. Why should requests use timeouts?

A. To prevent indefinite waiting

B. To authenticate users

C. To convert JSON

D. To create databases

Answer: A

6. What does `raise_for_status()` do?

A. Creates an HTTP request

B. Raises an exception for unsuccessful HTTP status codes

C. Parses JSON

D. Adds authentication

Answer: B

7. Which status code commonly means "Not Found"?

A. 200

B. 201

C. 404

D. 500

Answer: C

8. What should you use instead of hard-coding API secrets?

A. Comments

B. Environment or secret configuration

C. HTML

D. Function names

Answer: B

---

57. Interview Questions

Beginner

1. How do you make a GET request in Python?
2. What is the `requests` library?
3. What is a response object?
4. How do you read JSON from a response?
5. How do you send query parameters?
6. How do you send JSON in a POST request?
7. What is the difference between `params=` and `json=`?
8. What is `response.status_code`?
9. What does `raise_for_status()` do?
10. Why are timeouts important?

Intermediate

11. How do you handle network errors?
12. How do you handle an invalid JSON response?
13. What is the purpose of `requests.Session()`?
14. How should API keys be stored?
15. What is the difference between PUT and PATCH?
16. What is API rate limiting?
17. What is exponential backoff?
18. What is idempotency?
19. Why can automatic retries be dangerous?
20. How would you design a reusable API client?

Practical

21. How would you handle a 401 response?
22. How would you handle a 429 response?
23. How would you handle a 500 response?
24. How would you test an API client without making real requests?
25. How would you prevent credentials from appearing in logs?
26. How would you structure a Python application that consumes multiple APIs?
27. When would you use `httpx` instead of `requests`?
28. How would you safely retry a network request?

---

58. Cheat Sheet

```
Install
pip install requests

Import
import requests

GET
requests.get(url)

POST
requests.post(url, json=data)

PUT
requests.put(url, json=data)

PATCH
requests.patch(url, json=data)

DELETE
requests.delete(url)

Query Parameters
requests.get(url, params=params)

Headers
requests.get(url, headers=headers)

JSON Body
requests.post(url, json=data)

Form Data
requests.post(url, data=data)

Timeout
requests.get(url, timeout=5)

Status Code
response.status_code

Final URL
response.url

Headers
```

```
response.headers

Text
response.text

Raw Bytes
response.content

JSON
response.json()

Check HTTP Errors
response.raise_for_status()

General Request Error
requests.RequestException

Session
requests.Session()

Environment Variable
os.getenv("API_KEY")

Common Success
200 OK
201 Created
204 No Content

Common Errors
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
429 Too Many Requests
500 Internal Server Error
503 Service Unavailable
```

---

59. Key Takeaways

1. `requests` provides a straightforward way to make HTTP requests from Python.
2. Use `GET` to retrieve data and other HTTP methods according to the API's contract.
3. Use `params=` for query parameters.

4. Use `json=` for JSON request bodies.
5. Use `headers=` for headers such as authentication and content negotiation.
6. Always consider a timeout for network requests.
7. Check HTTP status codes before treating a response as successful.
8. Use `raise_for_status()` when appropriate.
9. Use `response.json()` only when JSON is expected.
10. Keep API keys and tokens out of source code and logs.
11. Handle network failures separately from HTTP errors when useful.
12. Be careful with automatic retries, especially for non-idempotent operations.
13. Use sessions when multiple requests share configuration or state.
14. Separate API client code from business logic as applications grow.
15. API documentation is the contract that tells your Python program how to communicate with the service.

Visit haas.dev for more resources and guides.

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