Building a Flask API

1. What Is a Flask API?

An API allows one program to communicate with another program.

A Flask API is a backend application built with Flask that receives HTTP requests and sends data back, usually as JSON.

For example, a frontend might request:

```
GET /api/resources
```

The Flask API can respond:

```
[
  {
    "id": 1,
    "title": "Python Basics"
  },
  {
    "id": 2,
    "title": "Flask Fundamentals"
  }
]
```

The frontend does not need to know how the data was stored or processed. It only needs to understand the API's request and response format.

Mental model

```
Frontend / Mobile App
        ↓
   HTTP Request
        ↓
    Flask API
        ↓
  Python Logic
        ↓
 Database / Service
        ↓
    Flask API
        ↓
    JSON Response
```

```
        ↓
Frontend / Mobile App
```

Flask routes incoming URLs to Python view functions. By default, routes handle `GET`, while other HTTP methods can be explicitly registered. Flask can also automatically convert dictionaries and lists returned from a view into JSON responses.

---

2. API vs Website

A Flask application can return HTML:

```
@app.get("/")
def home():
    return "<h1>Hello</h1>"
```

An API normally returns structured data:

```
@app.get("/api/user")
def get_user():
    return {
        "id": 1,
        "name": "Hafsa"
    }
```

The difference is important.

Website

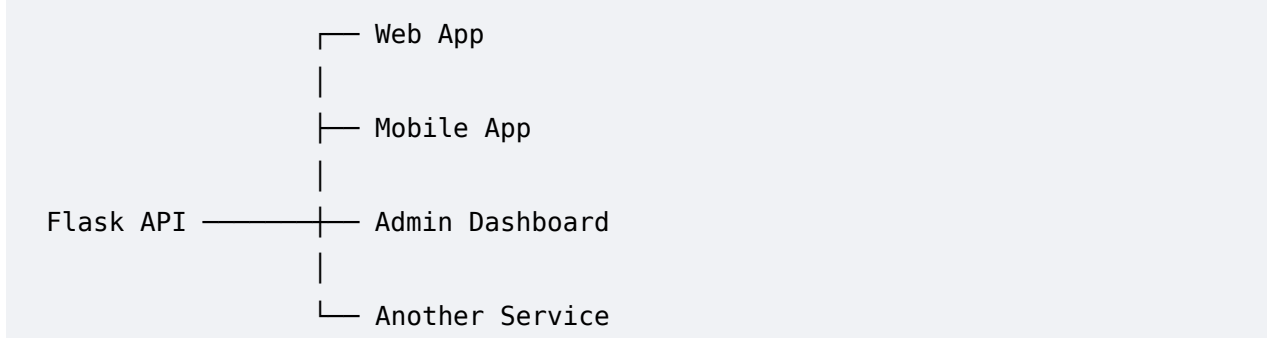
```
Browser → Flask → HTML → Browser
```

API

```
Frontend → Flask → JSON → Frontend
```

An API is particularly useful when multiple clients need the same backend.

For example:



---

3. What We Will Build

We will create a small **Resource API** similar to the kind of backend that could power a learning-resource platform.

Our API will support:

```
GET    /api/resources
GET    /api/resources/<id>
POST   /api/resources
PUT    /api/resources/<id>
DELETE /api/resources/<id>
```

This gives us the basic CRUD operations:

Operation	**HTTP Method**	**Purpose**
Create	POST	Add a resource
Read	GET	Retrieve resources
Update	PUT	Replace/update a resource
Delete	DELETE	Remove a resource

---

4. Create the Project

Create a folder:

```
flask-api/
```

A simple starting structure:

```
flask-api/
|
```

```
├── .venv/
├── app.py
└── requirements.txt
```

Create a virtual environment:

```
python -m venv .venv
```

Activate it.

Windows

```
.venv\Scripts\activate
```

macOS/Linux

```
source .venv/bin/activate
```

Install Flask:

```
pip install Flask
```

Save dependencies:

```
pip freeze > requirements.txt
```

---

5. Create the Flask Application

Create `app.py`:

```
from flask import Flask

app = Flask(__name__)


@app.get("/")
def home():
    return {
        "message": "Flask API is running"
    }


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

Run:

```
python app.py
```

You should see Flask running on a local address.

Open:

```
http://127.0.0.1:5000/
```

Response:

```
{
  "message": "Flask API is running"
}
```

Flask's application object acts as the central registry for configuration, routes, views, and other application components.

---

6. Understanding `@app.get()`

This:

```
@app.get("/api/resources")
def get_resources():
    ...
```

means:

> When a GET request arrives at `/api/resources`, execute `get_resources()`.

For example:

```
GET /api/resources
```

matches:

```
@app.get("/api/resources")
def get_resources():
```

Flask also supports the more general form:

```
@app.route("/api/resources", methods=["GET"])
```

And HTTP-method shortcuts such as:

```
@app.get(...)
@app.post(...)
@app.put(...)
@app.delete(...)
```

are available in modern Flask.

---

7. Returning JSON

An API normally communicates using JSON.

You can return a dictionary directly:

```
@app.get("/api/status")
def status():
    return {
        "status": "ok",
        "service": "resource-api"
    }
```

Or explicitly use `jsonify()`:

```
from flask import jsonify


@app.get("/api/status")
def status():
    return jsonify({
        "status": "ok",
        "service": "resource-api"
    })
```

Modern Flask automatically converts dictionaries and lists returned by view functions into JSON responses. `jsonify()` remains useful when you want explicit JSON response construction.

---

8. Create Our First API Endpoint

Start with some temporary data:

```
resources = [
    {
        "id": 1,
        "title": "Python Basics",
        "category": "Python"
    },
    {
        "id": 2,
        "title": "Flask Fundamentals",
        "category": "Python"
```

```
    }
]
```

Then:

```
@app.get("/api/resources")
def get_resources():
    return resources
```

Request:

```
GET /api/resources
```

Response:

```
[
  {
    "id": 1,
    "title": "Python Basics",
    "category": "Python"
  },
  {
    "id": 2,
    "title": "Flask Fundamentals",
    "category": "Python"
  }
]
```

---

9. API URLs and Resources

Good APIs usually organize URLs around **resources**.

Instead of:

```
/getAllResources
/createResource
/deleteResource
```

use:

```
/resources
```

and let the HTTP method communicate the action.

```
GET    /resources
POST   /resources
```

```
PUT    /resources/1
DELETE /resources/1
```

The URL identifies the resource.

The HTTP method identifies what you want to do with it.

---

10. Getting One Resource

We can use a dynamic URL:

```
@app.get("/api/resources/<int:resource_id>")
def get_resource(resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    return {
        "error": "Resource not found"
    }, 404
```

Now:

```
GET /api/resources/1
```

returns:

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

While:

```
GET /api/resources/99
```

returns:

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

with HTTP status:

```
404 Not Found
```

Flask supports typed URL converters such as `<int:resource_id>`, which causes the route parameter to be converted to an integer.

---

11. Understanding HTTP Status Codes

An API should communicate whether an operation succeeded.

Common codes:

Code	Meaning
200	Request succeeded
201	Resource created
204	Success with no response body
400	Invalid request
401	Authentication required/failed
403	Forbidden
404	Resource not found
409	Conflict
422	Validation problem

500	Server error

For example:

```
  return resource, 404
```

means:

```
  Response body: resource
  Status: 404
```

---

12. Using `request`

Flask provides the `request` object for accessing incoming request data.

```
  from flask import request
```

You can inspect:

```
  request.method
  request.args
  request.headers
  request.json
  request.form
```

The request object contains information about the incoming HTTP request, including its method, headers, path, query string, and other request data.

---

13. Reading Query Parameters

Suppose the client sends:

```
  GET /api/resources?category=Python
```

We can read `category`:

```
@app.get("/api/resources")
def get_resources():
    category = request.args.get("category")

    if category:
        filtered = [
```

```
            resource
            for resource in resources
            if resource["category"].lower() == category.lower()
        ]

        return filtered

    return resources
```

Now:

```
/api/resources
```

returns everything.

But:

```
/api/resources?category=Python
```

returns only Python resources.

---

14. Query Parameters vs URL Parameters

These are different.

URL parameter

```
/api/resources/5
```

Code:

```
@app.get("/api/resources/<int:resource_id>")
def get_resource(resource_id):
```

It identifies a specific resource.

Query parameter

```
/api/resources?category=Python
```

Code:

```
request.args.get("category")
```

It usually filters, searches, sorts, or modifies a request.

Think:

```
/resources/5
           ↑
```

```
       WHICH resource?

/resources?category=Python
            ↑
       FILTER resources
```

---

15. Creating Resources with POST

Now we need to accept data from the client.

Example request:

```
POST /api/resources
Content-Type: application/json
```

Body:

```
{
  "title": "Flask APIs",
  "category": "Python"
}
```

Flask code:

```
@app.post("/api/resources")
def create_resource():
    data = request.get_json()

    resource = {
        "id": len(resources) + 1,
        "title": data["title"],
        "category": data["category"]
    }

    resources.append(resource)

    return resource, 201
```

The flow is:

```
Client
  ↓
POST request
  ↓
JSON body
```

```
  ↓
request.get_json()
  ↓
Python dictionary
  ↓
Create object
  ↓
Store object
  ↓
Return JSON
```

---

16. Validate Incoming Data

Never assume the client sends correct data.

This is unsafe:

```
title = data["title"]
```

If `title` is missing, the application can fail.

Instead:

```
@app.post("/api/resources")
def create_resource():
    data = request.get_json(silent=True) or {}

    title = data.get("title")
    category = data.get("category")

    if not title or not category:
        return {
            "error": "title and category are required"
        }, 400

    resource = {
        "id": len(resources) + 1,
        "title": title,
        "category": category
    }

    resources.append(resource)

    return resource, 201
```

The API now handles invalid input deliberately.

---

17. Why Validation Matters

A client might send:

```
{}
```

or:

```
{
  "title": ""
}
```

or:

```
{
  "title": 123,
  "category": null
}
```

Your backend cannot trust incoming data.

The client could be:

- A browser
- A mobile app
- Another server
- A script
- A malicious user

The server must enforce its own rules.

---

18. Updating a Resource

A basic `PUT` endpoint:

```
@app.put("/api/resources/<int:resource_id>")
def update_resource(resource_id):
    data = request.get_json(silent=True) or {}

    for resource in resources:
        if resource["id"] == resource_id:
            resource["title"] = data.get(
                "title",
```

```
                resource["title"]
            )

            resource["category"] = data.get(
                "category",
                resource["category"]
            )

            return resource

    return {
        "error": "Resource not found"
    }, 404
```

Request:

```
PUT /api/resources/1
```

Body:

```
{
  "title": "Advanced Python"
}
```

The resource is updated.

---

19. PUT vs PATCH

Both can update resources, but they represent different ideas.

PUT

Usually represents replacing or updating the resource representation.

```
PUT /resources/1
```

PATCH

Usually represents a partial modification.

```
PATCH /resources/1
```

For example:

```
{
  "title": "Advanced Python"
}
```

only changes the title.

A real API should define its update semantics clearly and consistently.

---

20. Deleting a Resource

```
@app.delete("/api/resources/<int:resource_id>")
def delete_resource(resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            resources.remove(resource)

            return {
                "message": "Resource deleted"
            }

    return {
        "error": "Resource not found"
    }, 404
```

Request:

```
DELETE /api/resources/2
```

Response:

```
{
  "message": "Resource deleted"
}
```

---

21. A Complete Small Flask API

Here is our basic API in one file:

```
from flask import Flask, request

app = Flask(__name__)

resources = [
    {
        "id": 1,
        "title": "Python Basics",
        "category": "Python"
    },
```

```
    {
        "id": 2,
        "title": "Flask Fundamentals",
        "category": "Python"
    }
]


@app.get("/")
def home():
    return {
        "message": "Resource API is running"
    }


@app.get("/api/resources")
def get_resources():
    category = request.args.get("category")

    if category:
        return [
            resource
            for resource in resources
            if resource["category"].lower() == category.lower()
        ]

    return resources


@app.get("/api/resources/<int:resource_id>")
def get_resource(resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    return {
        "error": "Resource not found"
    }, 404


@app.post("/api/resources")
def create_resource():
    data = request.get_json(silent=True) or {}
```

```
    title = data.get("title")
    category = data.get("category")

    if not title or not category:
        return {
            "error": "title and category are required"
        }, 400

    resource = {
        "id": len(resources) + 1,
        "title": title,
        "category": category
    }

    resources.append(resource)

    return resource, 201


@app.put("/api/resources/<int:resource_id>")
def update_resource(resource_id):
    data = request.get_json(silent=True) or {}

    for resource in resources:
        if resource["id"] == resource_id:
            resource["title"] = data.get(
                "title",
                resource["title"]
            )

            resource["category"] = data.get(
                "category",
                resource["category"]
            )

            return resource

    return {
        "error": "Resource not found"
    }, 404
```

```
@app.delete("/api/resources/<int:resource_id>")
def delete_resource(resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            resources.remove(resource)

            return {
                "message": "Resource deleted"
            }

    return {
        "error": "Resource not found"
    }, 404


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

This is useful for learning, but it is **not production storage**. The list exists only in application memory.

Restarting the application resets the data.

---

22. Test the API

You can use:

- Browser
- `curl`
- Postman
- Insomnia
- VS Code REST Client
- Python scripts
- Automated tests

For example:

```
curl http://127.0.0.1:5000/api/resources
```

Create a resource:

```
curl -X POST http://127.0.0.1:5000/api/resources \
-H "Content-Type: application/json" \
-d "{\"title\":\"Flask APIs\",\"category\":\"Python\"}"
```

Get resource 1:

```
curl http://127.0.0.1:5000/api/resources/1
```

Delete resource 2:

```
curl -X DELETE http://127.0.0.1:5000/api/resources/2
```

---

23. API Clients

Once the API exists, another Python program can consume it.

Using `requests`:

```
import requests

response = requests.get(
    "http://127.0.0.1:5000/api/resources"
)

response.raise_for_status()

resources = response.json()

print(resources)
```

The architecture becomes:

```
Python Client
      ↓
HTTP Request
      ↓
Flask API
      ↓
Python Backend
      ↓
JSON Response
      ↓
Python Client
```

This is one of the most important concepts in backend development:

> An API creates a contract between the client and the server.

---

24. API Contract

Suppose your endpoint is:

```
GET /api/resources/1
```

The API contract might define:

Request

```
GET /api/resources/1
```

Success response

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

Not found

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

Status codes

```
200 → found
404 → not found
```

The frontend can now reliably understand what the backend means.

---

25. Consistent Error Responses

Instead of returning completely different structures:

```
{
  "error": "Not found"
}
```

and:

```
{
  "message": "Something went wrong"
}
```

you can establish a consistent format:

```
{
  "error": {
    "code": "RESOURCE_NOT_FOUND",
    "message": "Resource not found"
  }
}
```

Consistency makes APIs easier to consume.

---

26. Using `abort()`

Flask provides `abort()` for stopping a request with an HTTP error.

```
from flask import abort
```

Example:

```
@app.get("/api/resources/<int:resource_id>")
def get_resource(resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    abort(404, description="Resource not found")
```

You can then define an API-friendly error handler:

```
from flask import jsonify


@app.errorhandler(404)
def handle_not_found(error):
    return jsonify({
        "error": error.description
    }), 404
```

Flask supports application-level error handlers, and API applications can use them to return JSON rather than default HTML error pages.

---

27. Handling 400 Errors

```
@app.errorhandler(400)
def handle_bad_request(error):
    return {
```

```
        "error": "Bad request"
    }, 400
```

Now invalid client requests can have a consistent JSON response.

---

28. Handling 500 Errors

A `500` means something went wrong on the server.

For example:

```
Client
  ↓
Valid request
  ↓
Flask
  ↓
Database crashes
  ↓
500 Internal Server Error
```

You can define:

```
@app.errorhandler(500)
def handle_server_error(error):
    return {
        "error": "Internal server error"
    }, 500
```

Do not expose internal stack traces, database details, secrets, or implementation information to API clients.

Flask returns `500 Internal Server Error` for unhandled exceptions by default and supports custom handlers for controlled error responses.

---

29. Debug Mode

During development:

```
app.run(debug=True)
```

Debug mode provides useful development features such as automatic reloading and an interactive debugger for unhandled exceptions.

However:

```
Development
    ↓
debug=True
```

is different from:

```
Production
    ↓
debug=False
```

Never deploy a production application with Flask's development debugger enabled. Flask's documentation explicitly warns against enabling debug mode in production.

---

30. API Versioning

As APIs evolve, changing existing endpoints can break clients.

Instead of:

```
/api/resources
```

you may eventually use:

```
/api/v1/resources
```

Later:

```
/api/v2/resources
```

For example:

```
@app.get("/api/v1/resources")
def get_resources():
    return resources
```

Versioning becomes particularly useful when an API has external consumers.

---

31. Separating Routes From Business Logic

Putting everything into `app.py` works for a small learning project.

A larger application should separate responsibilities.

For example:

```
flask-api/
|
├── app/
```

```
│   ├── __init__.py
│   ├── routes/
│   │   └── resources.py
│   ├── services/
│   │   └── resource_service.py
│   ├── models/
│   │   └── resource.py
│   └── errors.py
│
├── tests/
│
├── config.py
├── requirements.txt
└── run.py
```

The exact structure can vary.

The important principle is:

```
Routes
  ↓
Services
  ↓
Data layer
```

A route should not become a giant function containing every piece of application logic.

---

32. What a Route Should Do

A route can coordinate the request:

```
@app.post("/api/resources")
def create_resource():
    data = request.get_json()

    resource = resource_service.create(data)

    return resource, 201
```

The service can handle business logic:

```
def create(data):
    validate_resource(data)

    resource = Resource(
```

```
        title=data["title"],
        category=data["category"]
    )

    return repository.save(resource)
```

This separation makes the application easier to test and maintain.

---

33. Blueprints

As the API grows, Flask Blueprints can organize related routes.

For example:

```
/api/resources/*
/api/users/*
/api/auth/*
```

could be separated into:

```
resources blueprint
users blueprint
auth blueprint
```

Example:

```
from flask import Blueprint

resources_bp = Blueprint(
    "resources",
    __name__,
    url_prefix="/api/resources"
)
```

Then:

```
@resources_bp.get("/")
def get_resources():
    ...
```

This allows larger applications to divide their URL and functionality into modules.

---

34. Database Integration

Our list:

```
resources = []
```

is temporary.

A real API usually uses a database:

```
Flask
  ↓
Service
  ↓
Repository / ORM
  ↓
Database
```

Possible databases include:

```
SQLite
PostgreSQL
MySQL
```

A typical request could look like:

```
GET /api/resources/10
        ↓
Flask route
        ↓
Resource service
        ↓
Database query
        ↓
Resource object
        ↓
JSON
```

Database integration should be taught separately because it introduces models, queries, transactions, migrations, relationships, connection management, and other concepts.

---

35. Authentication

A production API may need to identify users.

Common approaches include:

```
Session-based authentication
API keys
```

```
Token authentication
JWT-based authentication
OAuth 2.0 / OpenID Connect
```

For example:

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

The API can authenticate the request before allowing access to protected resources.

Authentication and authorization are different:

```
Authentication
"Who are you?"

Authorization
"What are you allowed to do?"
```

Do not confuse the two.

---

36. CORS

Suppose your frontend runs on:

```
http://localhost:3000
```

and your Flask API runs on:

```
http://localhost:5000
```

They have different origins.

Browser security rules can therefore affect requests between them.

Cross-Origin Resource Sharing, or CORS, controls which origins are allowed to access resources.

In a real application, configure CORS deliberately rather than allowing every origin by default.

---

37. API Security Basics

A Flask API should treat incoming requests as untrusted.

Important practices include:

Validate input

```
if not title:
    return {"error": "title is required"}, 400
```

Never expose secrets

Do not write:

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

in source code.

Use environment variables instead.

Use HTTPS

Production APIs should normally use encrypted HTTPS connections.

Apply authorization

Do not assume that because a user is authenticated, they can access every resource.

Set appropriate limits

Consider:

- Request size
- Timeouts
- Rate limits
- Pagination
- Upload limits

---

38. Pagination

Imagine your database contains:

```
100,000 resources
```

Returning everything from:

```
GET /api/resources
```

is inefficient.

Instead:

```
GET /api/resources?page=1&limit=20
```

Response:

```
{
  "data": [
    ...
  ],
  "page": 1,
  "limit": 20,
```

```
  "total": 100000
}
```

Pagination prevents APIs from unnecessarily returning huge datasets.

---

39. Filtering, Searching, and Sorting

A mature API may support:

```
/api/resources?category=Python
```

Filtering:

```
/api/resources?category=Python
```

Searching:

```
/api/resources?search=flask
```

Sorting:

```
/api/resources?sort=title
```

Pagination:

```
/api/resources?page=2&limit=20
```

These can also be combined:

```
/api/resources?category=Python&search=flask&page=2&limit=20
```

The API should define these parameters clearly.

---

40. Testing a Flask API

Do not rely only on manually opening URLs.

Flask provides testing support that can be used with testing frameworks such as pytest.

A simple test might look like:

```
def test_get_resources(client):
    response = client.get("/api/resources")

    assert response.status_code == 200
```

Testing can verify:

```
GET works
POST validates input
PUT updates correctly
DELETE removes correctly
404 responses work
400 responses work
```

For a serious API, automated tests should cover both successful and failure scenarios.

---

41. Common Flask API Mistakes

Mistake 1: Putting everything in one file

Small projects are fine.

Large projects become difficult to maintain.

Mistake 2: Trusting client input

Never assume:

```
{
  "title": "correct"
}
```

will always be sent.

Validate it.

Mistake 3: Returning everything

Avoid returning thousands of records without pagination.

Mistake 4: Ignoring status codes

This:

```
return {"error": "Not found"}
```

without a status code may return `200`.

Instead:

```
return {"error": "Not found"}, 404
```

Mistake 5: Exposing internal errors

Do not return:

```
Database password...
SQL query...
Stack trace...
File system path...
```

to clients.

Mistake 6: Using debug mode in production

```
debug=True
```

belongs in development.

Mistake 7: Mixing business logic into routes

Avoid:

```
@app.post("/api/orders")
def create_order():
    # 200 lines of validation,
    # calculations,
    # database queries,
    # email sending...
```

Keep routes focused.

Mistake 8: No API contract

Clients should know:

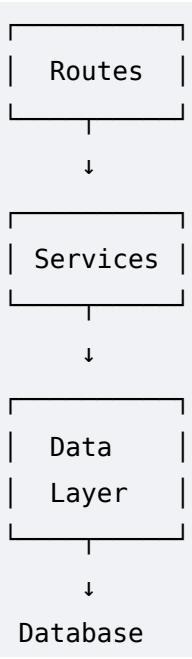
```
URL
Method
Input
Output
Status codes
Errors
```

---

42. Better API Design Mental Model

Think in layers:

```
              Client
                ↓
         HTTP Request
                ↓
```



And back:

```
Database
   ↓
Data layer
   ↓
Service
   ↓
Route
   ↓
JSON Response
   ↓
Client
```

This mental model becomes important when moving from small Flask projects to production backends.

---

43. Real-World haas.dev Example

Imagine haas.dev has a frontend that needs resource data.

The frontend requests:

```
GET /api/resources?category=Python
```

Flask receives it:

```
@app.get("/api/resources")
def get_resources():
```

```
    category = request.args.get("category")

    return resource_service.list(
        category=category
    )
```

The service handles the logic:

```
def list(category=None):
    resources = repository.get_all()

    if category:
        resources = [
            resource
            for resource in resources
            if resource.category == category
        ]

    return resources
```

The frontend receives:

```
[
  {
    "id": 1,
    "title": "Python Basics",
    "category": "Python"
  }
]
```

Now the frontend does not care how resources are stored.

That separation is the real value of an API.

---

44. Five-Minute Challenge

Build this endpoint:

```
GET /api/health
```

It should return:

```
{
  "status": "healthy",
  "service": "resource-api"
}
```

Then add:

```
GET /api/resources/<id>
```

Requirements:

- Return the resource if it exists.
- Return `404` if it does not.
- Return JSON.
- Use an integer URL converter.

---

45. Exercises

Exercise 1: Search

Create:

```
GET /api/resources?search=python
```

Return resources whose titles contain the search text.

Exercise 2: Validation

Modify the POST endpoint so that:

- `title` is required.
- `category` is required.
- Empty strings are rejected.

Exercise 3: PATCH

Create:

```
PATCH /api/resources/<id>
```

Allow the client to change only the fields it provides.

Exercise 4: Pagination

Implement:

```
GET /api/resources?page=1&limit=5
```

Exercise 5: Error Handler

Create a global JSON response for `404`:

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

---

46. Mini Project: Resource REST API

Build a Flask API for a learning platform.

Resources

Each resource should have:

```
id
title
description
category
level
```

Example:

```
{
  "id": 1,
  "title": "Flask Fundamentals",
  "description": "Learn the basics of Flask.",
  "category": "Python",
  "level": "Beginner"
}
```

Required endpoints

```
GET    /api/resources
GET    /api/resources/<id>
POST   /api/resources
PUT    /api/resources/<id>
PATCH  /api/resources/<id>
DELETE /api/resources/<id>
```

Add filtering

```
/api/resources?category=Python
```

```
/api/resources?level=Beginner
```

Add search

```
/api/resources?search=flask
```

Add error handling

Handle:

```
400
404
500
```

Then improve the architecture

Move from:

```
app.py
```

to:

```
routes
services
models
repository
tests
```

That progression teaches much more than simply creating endpoints.

---

47. Interview Questions

Beginner

1. What is a Flask API?
2. What is the difference between a website and an API?
3. What is a route?
4. What does `@app.get()` do?
5. What is JSON?
6. What is the purpose of `request`?
7. What are query parameters?
8. What is a URL parameter?
9. What does HTTP `404` mean?
10. Why is `201` commonly used after creating a resource?

Intermediate

11. What is the difference between PUT and PATCH?
12. Why should API input be validated?
13. What is API versioning?

14. What is pagination?
15. What is CORS?
16. What is the difference between authentication and authorization?
17. Why should business logic be separated from routes?
18. What are Flask Blueprints?
19. How should API errors be represented?
20. Why should debug mode not be enabled in production?

Advanced

21. How would you structure a large Flask API?
22. How would you implement centralized error handling?
23. How would you design API versioning?
24. How would you prevent an API from returning millions of records?
25. How would you test Flask API endpoints?
26. How would you handle authentication and authorization?
27. How would you protect sensitive configuration?
28. How would you introduce a database without putting database logic inside routes?
29. How would you design a consistent API response format?
30. How would you monitor and debug a Flask API in production?

---

48. API Cheat Sheet

Create application

```
from flask import Flask

app = Flask(__name__)
```

GET

```
@app.get("/api/resources")
def get_resources():
    return resources
```

POST

```
@app.post("/api/resources")
def create_resource():
    data = request.get_json()
    return data, 201
```

PUT

```
@app.put("/api/resources/<int:id>")
def update_resource(id):
    ...
```

PATCH

```
@app.patch("/api/resources/<int:id>")
def patch_resource(id):
    ...
```

DELETE

```
@app.delete("/api/resources/<int:id>")
def delete_resource(id):
    ...
```

Query parameter

```
request.args.get("category")
```

JSON body

```
request.get_json()
```

Status code

```
return {"message": "Created"}, 201
```

Error

```
return {"error": "Not found"}, 404
```

Abort

```
abort(404)
```

Error handler

```
@app.errorhandler(404)
def not_found(error):
    return {"error": "Not found"}, 404
```

---

49. Key Takeaways

- Flask can be used to build Python APIs.
- An API receives HTTP requests and returns responses.
- API responses are commonly JSON.
- Routes connect URLs and HTTP methods to Python functions.
- `GET` reads data.
- `POST` creates data.
- `PUT` updates/replaces data.
- `PATCH` partially updates data.
- `DELETE` removes data.
- `request` provides access to incoming request data.
- Query parameters are useful for filtering, searching, sorting, and pagination.
- URL parameters commonly identify individual resources.
- HTTP status codes communicate the result of an operation.
- API input must be validated.
- Error responses should be consistent and useful.
- Business logic should eventually be separated from route functions.
- Blueprints help organize larger Flask applications.
- Real APIs normally use persistent databases rather than Python lists.
- Authentication and authorization solve different problems.
- Debug mode is for development, not production.
- Testing should cover both successful requests and failures.
- A good API is not just a collection of endpoints; it is a predictable contract between clients and servers.

Core mental model:

```
HTTP Request
     ↓
   Flask
     ↓
   Route
     ↓
  Service
     ↓
 Data Layer
     ↓
 Database
```

```
      ↓
  JSON Response
      ↓
    Client
```

Remember:

> Build the API around resources, make the contract predictable, validate everything coming from the client, and keep business logic separate from HTTP handling.

Visit haas.dev for more resources and guides.

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