

REST principles & constraints — a deep, practical guide

REST is more than “HTTP + JSON”. It’s an architectural style defined by Roy Fielding in his dissertation. When done thoughtfully, RESTful design gives you APIs that are scalable, evolvable, easy to cache, and intuitive for clients. Below I’ll walk through the **constraints that make REST**, what each one means in practice, why it matters, and how you — as a Flask developer / trainer — should apply them when designing APIs. No code samples — just clear concepts and actionable guidance.

Quick orientation: the goal of REST

REST (Representational State Transfer) aims to make distributed systems simple and scalable by prescribing **how components interact** (clients, servers, intermediaries). Following REST constraints improves reusability, visibility, reliability and scalability.

The six core constraints (plus the optional one)

1. **Client–Server**
2. **Statelessness**
3. **Cacheable**
4. **Uniform Interface** (this one contains 4 sub-constraints)
5. **Layered System**
6. **Code on Demand** (optional)

We’ll unpack each and then cover practical design rules, pitfalls, and a checklist for Flask-based APIs.

1) Client–Server

Idea: separate concerns — client handles UI/user state, server handles data storage & business logic.

Why it matters: separation enables independent evolution: you can change your server internals (DB, framework) without breaking clients.

Practical notes for Flask apps

- Keep UI-specific logic out of your Flask API code. Build thin, focused APIs that return resource representations.
- Use JSON for data interchange; let clients transform/display it as needed.

2) Statelessness

Idea: each request from client to server must contain *all* information to understand and process it. The server does **not** keep client session state between requests.

Why it matters: stateless servers scale horizontally more easily (any request can go to any host), failures are simpler to isolate, and caching behaves well.

Practical application

- Use tokens (JWT, opaque bearer tokens) or API keys sent in `Authorization` headers for authentication. Don't rely on server-side session memory for API requests.
- If some state must be tracked, store it on the client or in a shared store (DB, Redis) referenced by a token.
- Make all requests self-descriptive (include content-type, auth headers, pagination params, etc.).

Common trap: storing significant per-client session data in-memory on a single server — breaks scaling and load balancing.

3) Cacheable

Idea: responses must declare themselves cacheable or non-cacheable. Well-chosen caching drastically improves performance and scalability.

Why it matters: reduces server load and latency — intermediaries (CDNs, proxies) can serve repeated requests.

Practical application

- Use proper HTTP caching headers on responses:
 - `Cache-Control` (public/private, max-age),
 - `Expires` when applicable,
 - `ETag` and `Last-Modified` with conditional requests (`If-None-Match`, `If-Modified-Since`) for efficient revalidation.
- Default: treat modifying operations (POST/PUT/PATCH/DELETE) as non-cacheable. GET responses can be cacheable if they are safe to cache.
- Patch or include `Vary` header when responses vary by `Accept` or `Authorization`.

Flask tip: use `Flask-Caching` or set headers explicitly in responses. Implement ETag generation for resource representations.

4) Uniform Interface (the heart of REST)

This is actually four constraints bundled under “uniform interface”:

a) Resource identification in requests

- **Resources** (things) must be identified in URLs. Use nouns — not verbs.
 - Good: `/api/customers/123`, `/api/orders/2023-10-01`
 - Avoid: `/api/getCustomer?id=123` as the primary pattern.

b) Resource manipulation through representations

- Clients fetch a **representation** (e.g., JSON) of a resource, modify it, then send it back to update the resource.
- Representations include media type (`Content-Type`) and possibly hypermedia links.

c) Self-descriptive messages

- Every message contains enough information to describe how to process it (headers, content-type, links).
- Use standard HTTP methods and status codes so behavior is predictable.

d) HATEOAS — Hypermedia As The Engine Of Application State

- Responses include links/actions so clients can discover next steps (e.g., `{"id":1, "name":"X", "links":[{"rel":"orders", "href":"/api/customers/1/orders"}]}`).
- **Practical caveat:** HATEOAS is ideal but many public APIs limit hypermedia in practice. Use HATEOAS or at least include actionable links for resources where flows are non-trivial (paginated collections, related resources, next/prev).

5) Layered System

Idea: the client should not need to know whether it's talking directly to the server or to an intermediary (proxy, gateway, cache).

Why it matters: enables load-balancers, API gateways, WAFs, CDNs to be inserted without changing client behavior.

Practical application

- Design APIs assuming a gateway (Kong) or reverse proxy might be present.
- Use standard headers for forwarding (`X-Forwarded-For`, etc.) when needed (but beware of trust boundaries).

6) Code on Demand (optional)

Idea: servers can send executable code (e.g., JavaScript) to extend client functionality temporarily. Rarely used for Web APIs — optional.

Richardson Maturity Model (how RESTful is your API?)

A useful framework to measure RESTfulness:

- **Level 0:** Single endpoint (RPC-over-HTTP).
- **Level 1:** Resources (distinct URIs for resources).
- **Level 2:** Uses HTTP verbs properly (GET/POST/PUT/DELETE, status codes).
- **Level 3:** HATEOAS — responses include hypermedia links.

Aim for Level 2 for practical, maintainable APIs. Level 3 is ideal for discoverability but costs more design effort.

Practical design guidance (Flask-focused, non-code)

Resource modeling & URIs

- Model resources as nouns and use plural resource names: `/api/products`, `/api/customers`.
- Use hierarchical relationships sparingly and only when meaningful: `/api/customers/123/orders`.
- Prefer unique identifiers in URIs (`/api/customers/{id}`); avoid exposing internal DB keys if sensitive.

HTTP method semantics

- **GET:** safe, idempotent — fetch a resource.
- **HEAD:** like GET but retrieves only headers (useful for caching).
- **POST:** create subordinate resources or perform non-idempotent operations.
- **PUT:** idempotent update/replace a resource.
- **PATCH:** partial update (must be deterministic).
- **DELETE:** idempotent removal.

Idempotency & safety

- Idempotent methods (PUT, DELETE) should behave the same when repeated. This matters for retries and network failures.
- POST is not idempotent by default — design accordingly (e.g., clients should not retry blindly).

Status codes — be precise

- 200 OK — success with response body
- 201 Created — resource created; include `Location` header pointing to new resource
- 204 No Content — success with no body (good for DELETE or successful PUT when no content returned)
- 400 Bad Request — client error / validation
- 401 Unauthorized — missing/invalid auth
- 403 Forbidden — authenticated but forbidden
- 404 Not Found — resource not present
- 409 Conflict — version collisions or business conflicts
- 422 Unprocessable Entity — validation problems (useful alternative to 400)
- 5xx — server errors

Content negotiation

- Use `Accept` and `Content-Type` headers. Support common JSON media types (`application/json`).
- Consider supporting `application/vnd.myapp.v1+json` for versioned media types if you follow media-type versioning.

Versioning & evolution

- Prefer versioning in the URI (`/api/v1/customers`) or via `Accept` header. Keep the strategy consistent.
- Use backward-compatible changes where possible; announce breaking changes and deprecate gracefully.

Pagination, filtering, sorting

- Provide robust pagination for collections:
 - Offset/limit (e.g., `?page=2&per_page=20`) or
 - Cursor-based for large/real-time sets (safer for inserts/deletes).
- Return pagination metadata and links (next/prev) in the response or headers (`Link` header).

Partial responses & sparse fieldsets

- Allow clients to request only needed fields (e.g., `?fields=id,name,email`), reducing bandwidth.

Rate limiting & throttling

- Apply rate limits (via gateway like Kong). Return appropriate headers: `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `Retry-After`.

Security essentials

- Always use TLS (HTTPS).
- Prefer OAuth2 / JWT for auth; use `Authorization: Bearer <token>`.
- Validate and sanitize all inputs. Use parameterized DB queries (avoid SQL injection).
- Use principle of least privilege for tokens and API keys.
- Avoid exposing sensitive implementation details in error responses.

Observability & reliability

- Add correlation IDs for tracing across services (pass via headers).
- Log structured JSON with request/response metadata and timing.
- Expose health-check endpoints for orchestration and load balancers.

Documentation & discoverability

- Publish OpenAPI/Swagger docs describing endpoints, payloads, and examples.
- Provide client SDKs or code samples if helpful.

HATEOAS: pros & cons

- **Pros:** clients can discover actions; reduces need for out-of-band documentation for flows.
- **Cons:** more design overhead, many clients and developers prefer explicit API docs and versioning.
- **Practical compromise:** include useful links for resources and pagination (`self`, `next`, `prev`, related resources) while relying on OpenAPI for detailed contracts.

Anti-patterns to avoid

- Using verbs in URLs (e.g., `/getUser`) — makes APIs RPC-like.
- Encoding too much state in URLs or using custom headers for conventional things.
- Returning 200 for all errors (obscures issues); prefer meaningful status codes.
- Creating deeply nested URLs just to represent relationships — think whether flattening + query param is better.
- Relying on server sessions for API client state.

Testing, CI, & evolution

- Use contract testing (Pact) when multiple teams own client & server.
- Apply automated tests (unit + integration) and run them in CI.
- Maintain a versioned OpenAPI spec; generate server/client stubs or docs from it.
- Use feature flags and blue/green or canary deployments for risk mitigation.

Quick checklist you can hand students (or use during design)

- ☐ Are my resources nouns and URIs stable?
- ☐ Are HTTP methods used according to semantics?
- ☐ Do all responses include correct status codes?
- ☐ Are GET responses cacheable where appropriate? ETag/Last-Modified in place?
- ☐ Is the API stateless (auth via tokens, no server sessions)?
- ☐ Are write operations idempotent where they should be?
- ☐ Is pagination implemented for collections with links/meta?
- ☐ Are error responses consistent and helpful (structured error object)?
- ☐ Is TLS enforced and auth implemented securely?
- ☐ Is OpenAPI documentation available and up to date?
- ☐ Have you considered rate limiting, logging, and monitoring?

Final thoughts (for Flask trainers)

- REST isn't a checklist you finish once — it's a set of practices that guide API design.
- For Flask lessons focus on: modeling resources, correct use of HTTP verbs/status codes, stateless auth (JWT), caching headers, and how to document with OpenAPI.
- Give learners exercises: move an RPC-style API to RESTful resources, add caching headers and ETags, and document the API with Swagger.

HTTP Methods & Status Codes — The Language of REST APIs

When we build RESTful APIs (whether in Flask or any other framework), **HTTP is the foundation**. REST isn't just about JSON over HTTP — it's about **using HTTP properly**: the **methods** (verbs) and **status codes** (responses).

If you get these right, your API feels intuitive, consistent, and professional. If you get them wrong, clients will struggle to consume and maintain it.

Why methods & status codes matter


- HTTP methods express the **intention** of the request.
- HTTP status codes express the **outcome** of the request.

Together, they create a universal contract between client and server — independent of implementation language.


The Core HTTP Methods for REST

There are many methods in the HTTP spec (OPTIONS, TRACE, CONNECT...), but in REST APIs we mostly rely on these six:


1. GET — Retrieve data

- **Semantics:** Read-only. Should not change server state.
- **Idempotent:**  Yes. Multiple GETs yield the same result (unless resource itself changes).
- **Use cases:**
 - Fetch a single resource: `/api/customers/42`
 - Fetch a collection: `/api/products?page=2`


2. POST — Create data / perform actions

- **Semantics:** Create a subordinate resource, or trigger a server-side action.
- **Idempotent:**  No. Two identical POSTs may create two different resources.
- **Use cases:**
 - Add a new customer: `POST /api/customers`
 - Submit a form or process a payment.

3. PUT — Replace a resource


- **Semantics:** Full replacement of an existing resource. If resource doesn't exist, some APIs allow creation.
- **Idempotent:**  Yes. Sending the same PUT repeatedly yields the same result.
- **Use cases:**
 - Update customer details completely: `PUT /api/customers/42`

4. PATCH — Update part of a resource

- **Semantics:** Partial update of fields.
- **Idempotent:**  Usually yes, but depends on implementation.
- **Use cases:**
 - Change only a customer's phone number: `PATCH /api/customers/42`

5. DELETE — Remove a resource













- **Semantics:** Delete a resource.

- **Idempotent:**  Yes. Deleting a resource multiple times results in same state (resource gone).
- **Use cases:**
 - Delete a customer: `DELETE /api/customers/42`

6. HEAD — Get metadata only

- **Semantics:** Same as GET, but returns only headers (no body).
- **Use cases:**
 - Check if resource exists.
 - Validate caching (via `ETag` or `Last-Modified`).

Quick Table: REST Method Semantics

Method	Safe (no side effects)?	Idempotent?	Common Use
GET	 Yes	 Yes	Retrieve resource(s)
POST	 No	 No	Create new resource, trigger action
PUT	 No (changes data)	 Yes	Replace resource
PATCH	 No	 Usually	Partial update
DELETE	 No	 Yes	Remove resource
HEAD	 Yes	 Yes	Retrieve headers/metadata

HTTP Status Codes — The Server's Reply

Think of status codes as the **API's body language**. They tell the client what happened. Grouped by category:

1xx — Informational

Rarely used in APIs. Example: `100 Continue`.

2xx — Success

- **200 OK** — Generic success (GET, PUT, PATCH).
- **201 Created** — Resource created successfully (use with POST; include `Location` header).
- **202 Accepted** — Request accepted but processing async (common in long-running jobs).

- **204 No Content** — Success with no body (DELETE, PUT when no response needed).

3xx — Redirection

- Rare in APIs, but **304 Not Modified** is important:
 - Used with caching (**ETag**, **If-None-Match**).
 - Tells client: "You already have the latest version."

4xx — Client Errors

These indicate a problem with the request.

- **400 Bad Request** — Invalid syntax or malformed request.
- **401 Unauthorized** — Missing/invalid authentication.
- **403 Forbidden** — Authenticated but not allowed.
- **404 Not Found** — Resource doesn't exist.
- **405 Method Not Allowed** — Method not supported on that resource.
- **409 Conflict** — Business conflict (e.g., booking same seat twice).
- **422 Unprocessable Entity** — Validation failed (useful for semantic errors).

5xx — Server Errors

These indicate the server failed to fulfill a valid request.

- **500 Internal Server Error** — Generic failure.
- **502 Bad Gateway** — Gateway (e.g., Kong) got invalid response from upstream.
- **503 Service Unavailable** — Service temporarily down/overloaded.
- **504 Gateway Timeout** — Upstream didn't respond in time.

Mapping Methods & Status Codes Together

- **GET** → 200 (OK), 404 (Not Found), 304 (Not Modified)
- **POST** → 201 (Created), 400 (Bad Request), 409 (Conflict)
- **PUT** → 200 (OK), 204 (No Content), 404 (Not Found)
- **PATCH** → 200 (OK), 204 (No Content), 404 (Not Found)
- **DELETE** → 204 (No Content), 404 (Not Found)

Practical Tips (for your Flask learners)

- Always return the **most specific** status code possible.
- Include helpful error bodies with 4xx responses (e.g., `{ "error": "Invalid email" }`).
- Use **Location** header when creating resources with POST.

- Respect idempotency — design PUT/DELETE so they can be retried safely.
- For long-running tasks, use **202 Accepted** + a polling endpoint.
- Avoid overloading 200 OK for everything — distinguish errors properly.

Summary

- HTTP **methods** define **what the client wants** (CRUD mapped to verbs).
- HTTP **status codes** define **what actually happened** (outcome).
- Used consistently, they make APIs predictable and self-explanatory.

In your Flask training, once students know these, they'll better understand why we use different route handlers (**GET**, **POST**, etc.) and why setting the right response code is just as important as returning JSON.

URI Design Best Practices for REST APIs

In RESTful API design, the **URI (Uniform Resource Identifier)** is more than just a path. It is the **address of a resource** — the way clients navigate and interact with your system.

Poorly designed URIs lead to confusion, inconsistencies, and maintenance issues. Well-designed URIs, on the other hand, make an API **predictable, intuitive, and easy to use**.

Key Principles of URI Design

1. Use Nouns, Not Verbs

- URIs represent **resources**, not actions.
- Actions should be communicated via HTTP methods (GET, POST, PUT, DELETE), not inside the URI.

✅ Good:

```
/customers/42  
/orders/123/items
```

❌ Bad:

```
/getCustomer?id=42  
/createOrder
```

2. Keep URIs Simple & Predictable

- A client should be able to **guess** the URI structure by understanding the resource model.
- Avoid unnecessary complexity or deep nesting.

✓ Good:

```
/products
/products/101
/products/101/reviews
```

✗ Bad:

```
/getAllProducts
/store/products/details/101
```

3. Use Plural Nouns for Collections

- Collections = plural, single resource = singular (by its identifier).
- Makes it consistent and aligns with natural language.

✓ Good:

```
/customers      → collection
/customers/42   → single resource
```

✗ Bad:

```
/customer
/customer?id=42
```

4. Hierarchical Relationships with Sub-Resources

- If a resource is logically contained within another, use nested URIs.

✓ Good:

```
/customers/42/orders
/orders/123/items
```

✗ Bad:

```
/getOrdersByCustomerId/42
/orderItemsForOrder123
```

5. Use Path for Resources, Query for Filtering

- Use **path parameters** to identify a resource.
- Use **query parameters** for filtering, sorting, and pagination.

✓ Good:

```
/products/101  
/products?category=electronics&sort=price&limit=10
```

✗ Bad:

```
/getProduct?id=101  
/products/category/electronics/sort/price
```

6. Use Consistent Naming Conventions

- Stick to **lowercase letters** and **hyphens** (–) to separate words.
- Avoid camelCase or underscores.

✓ Good:

```
/customer-orders  
/product-categories
```

✗ Bad:

```
/CustomerOrders  
/product_categories
```

7. Avoid Deep Nesting

- Too much nesting makes URIs hard to manage.
- Limit depth to 2–3 levels; use query parameters or links for complex relationships.

✓ Good:

```
/customers/42/orders
```

✗ Bad:

```
/customers/42/orders/123/items/5/payments/999
```

8. Use HTTP Methods Instead of Action Words

- Don't encode actions like `update`, `delete`, or `create` into the URI.
- HTTP methods already carry that meaning.

✓ Good:

```
DELETE /customers/42
```

✗ Bad:

```
/customers/42/delete
```

9. Support Pagination & Filtering

- Collections can grow large; support query parameters for pagination.

✓ Example:

```
/products?page=2&limit=20  
/customers?city=delhi
```

10. Version Your API in the URI (or headers)

- APIs evolve; versioning prevents breaking clients.
- Version is usually placed at the start of the path.

✓ Good:







```
/v1/customers  
/v2/customers
```

✗ Bad:

```
/customers?version=1
```

URI Design Checklist

Before finalizing your API URIs, ask:

-  Does it use **nouns**, not verbs?
-  Is it **predictable** and **consistent**?
-  Are resources properly **pluralized**?
-  Is **nesting kept minimal**?
-  Are **query params used for filtering/pagination**?
-  Is **versioning handled** cleanly?

Summary

Well-designed URIs make your API:

- **Intuitive** (easy to learn by looking at a few examples).
- **Consistent** (same patterns apply across resources).
- **Future-proof** (easy to extend with filters, versions, new resources).

In your Flask projects, always design URIs before coding. Think of them as the **contract** between your backend and the world.

Flask Framework: Introduction & Philosophy

When learning to build web applications and REST APIs in Python, **Flask** often comes up as one of the most popular frameworks. But why Flask? What makes it special compared to other frameworks? And what philosophy drives its design?

This post introduces Flask, explains its philosophy, and sets the stage for why it's an excellent choice for learning (and building) RESTful APIs.

What is Flask?

Flask is a **lightweight, micro web framework** for Python.

- **Created in 2010** by Armin Ronacher as an experiment, it quickly grew into one of the most widely used Python frameworks for web development.
- Flask provides the **essentials** for building a web application (routing, request handling, templates), but leaves advanced features (authentication, database layers, form validation) up to extensions.
- It follows the **WSGI (Web Server Gateway Interface)** standard, meaning it can work with a variety of servers and environments.

Flask in One Line

Flask is a “**micro but extensible**” framework: small in core, but flexible enough to grow with your project.

Why Flask?

1. Simplicity

- Minimalistic by design.
- You can get a web app running in just a few lines of code.
- Perfect for beginners who want to understand *how the web works*.

2. Flexibility

- No “one-size-fits-all” rules.
- You decide how to structure your project, what database to use, and what libraries to integrate.

3. Extensibility

- A huge ecosystem of **Flask extensions** exists (Flask-SQLAlchemy, Flask-Login, Flask-RESTful, etc.).
- You can plug in just the parts you need.

4. Community Support

- Flask has a massive, active community.
- Tons of tutorials, guides, and open-source projects.

5. Great for APIs

- Since it’s lightweight and doesn’t enforce HTML templates, Flask is ideal for **building REST APIs** where you just send/receive JSON.

Flask’s Core Philosophy

Flask was designed around a few guiding principles that shaped its identity:

1. Minimalism

- Provide just the essentials: routing, request/response handling.
- Everything else is optional.
- This keeps Flask **lean and unopinionated**.

2. Explicit is Better than Implicit

- Following Python’s philosophy (PEP 20), Flask avoids “magic.”
- What you see in your code is what happens — making it easier to debug and learn.

3. Flexibility over Convention

- Unlike frameworks such as Django, Flask does not force you into a particular project structure or ORM.
- You decide how to organize your codebase.

4. Extensibility

- Need authentication? Add **Flask-Login**.
- Need a database? Add **Flask-SQLAlchemy**.
- Need an API toolkit? Add **Flask-RESTful** or **Flask-JWT-Extended**.
- Flask grows with you, instead of overwhelming you at the start.

5. Simplicity for Learning

- Because Flask is small and explicit, it is a fantastic **teaching tool**.
- You learn core web concepts: requests, responses, routing, middleware — without a huge abstraction layer in between.

Flask vs Other Frameworks

- **Django**: “Batteries-included.” Comes with ORM, authentication, admin panel. Opinionated. Great for big applications with standard patterns.
- **FastAPI**: Modern, async-first, with built-in validation and OpenAPI docs. Great for new projects focused on performance.
- **Flask**: Minimal, flexible, mature. You build only what you need, piece by piece.

👉 Think of Flask as **LEGO blocks**: you get a starter kit, and you choose what to build on top of it.

Summary

- Flask is a **micro, extensible framework** for Python.
- Its philosophy emphasizes **simplicity, flexibility, and explicitness**.
- Unlike heavier frameworks, Flask doesn’t dictate how you build — you decide.
- This makes it a perfect fit for learning **REST API design** and building real-world microservices.

In short: 👉 If you want to understand the **foundations of web APIs**, Flask is one of the best starting points.

Flask Application Structure & Configuration

When starting with Flask, it's tempting to keep everything inside a single `app.py` file. While that's fine for quick demos, real-world APIs need **organization and configurability**.

In this post, we'll explore:

- Common **Flask project structures**
- How to manage **configuration settings**
- Best practices to keep apps **scalable and maintainable**

1. Why Application Structure Matters

A simple `app.py` works well for a "Hello, World!" project:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, Flask!"
```

But as soon as you add:

- Multiple routes (e.g., `/api/products`, `/api/customers`)
- Database models
- Blueprints for modularization
- Middleware (CORS, authentication, logging)

...it becomes **unmanageable**.

👉 A **structured project layout** helps keep code **organized, testable, and scalable**.

2. Common Flask Application Layouts

(a) Single File Layout (for small apps)

```
myapp/
└─ app.py
```

✅ Good for prototyping, but not scalable.

(b) Package Layout (recommended for APIs)

```
myapp/
├── app/
│   ├── __init__.py    # App factory, initialization
│   ├── routes.py      # Routes / controllers
│   ├── models.py      # Database models
│   ├── config.py      # Configuration classes
│   ├── extensions.py  # Third-party extensions (db, jwt, etc.)
│   └── utils.py       # Helper functions
├── tests/             # Unit & integration tests
├── migrations/        # DB migrations (if using SQLAlchemy)
├── requirements.txt
└── wsgi.py            # Entry point for running with Gunicorn
```

👉 This layout is **modular** and is widely used for production APIs.

3. The Application Factory Pattern

Instead of creating the Flask app directly in `app.py`, use an **application factory**:

`app/__init__.py`

```
from flask import Flask

def create_app(config_class="app.config.DevConfig"):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Register blueprints here
    from app.routes import api_bp
    app.register_blueprint(api_bp, url_prefix="/api")

    return app
```

- `create_app()` builds the Flask app dynamically.
- Makes testing easier (you can spin up multiple instances with different configs).
- Keeps concerns separate.

4. Flask Configuration Management

Every application needs **settings** like:

- Debug mode
- Database connection URL
- JWT secret key

- Logging settings
- API keys

Instead of hardcoding them, Flask provides a clean way to manage configurations.

(a) Configuration Classes

```
class BaseConfig:
    SECRET_KEY = "default-secret"
    DEBUG = False
    TESTING = False

class DevConfig(BaseConfig):
    DEBUG = True
    DATABASE_URI = "sqlite:///dev.db"

class TestConfig(BaseConfig):
    TESTING = True
    DATABASE_URI = "sqlite:///test.db"

class ProdConfig(BaseConfig):
    DATABASE_URI = "postgresql://user:password@db/prod"
```

- Use inheritance so you don't repeat common values.
- Keep sensitive values (e.g., passwords) **outside source code** (use environment variables).

(b) Loading Configs

In `__init__.py`:

```
app.config.from_object("app.config.DevConfig")
```

Or load from environment:

```
import os
env_config = os.getenv("FLASK_CONFIG", "app.config.DevConfig")
app.config.from_object(env_config)
```

👉 This allows switching configs easily:

```
$ export FLASK_CONFIG=app.config.ProdConfig
```

5. Best Practices for Config

- **Don't hardcode secrets** → use `.env` files with libraries like `python-dotenv`.
- **Separate configs by environment** (Dev, Test, Prod).
- **Use application factory** for flexibility.
- **Store extensions separately** in `extensions.py` (SQLAlchemy, JWT, CORS, etc.).

6. Summary

- **Flat single-file apps** are fine for small demos, but for real APIs, use a **package structure**.
- Adopt the **application factory pattern** for flexibility and testing.
- Manage configuration using **classes** and environment variables.
- Follow best practices to keep your API **secure, scalable, and maintainable**.

✅ With this structure in place, your Flask project is ready to scale from a toy app → to a **production-ready API**. The next natural step is: **"Flask Routing & Blueprints"** — where we modularize our endpoints.

JSON Request/Response Handling in Flask

When building modern web applications and APIs, JSON (JavaScript Object Notation) is the most common format for communication between client and server. Flask, a lightweight Python web framework, makes working with JSON requests and responses simple and intuitive.

In this post, we'll walk through how to handle JSON in Flask—covering both incoming requests and outgoing responses—with clear examples.

Why JSON?

JSON is human-readable, lightweight, and supported across platforms and programming languages. It's the backbone of RESTful APIs and microservices because of its simplicity and interoperability.

Setting Up a Basic Flask Application

Before diving into JSON handling, let's set up a simple Flask app.

```
from flask import Flask, request, jsonify

app = Flask(__name__)
```

Handling JSON Requests

When a client sends data in JSON format (usually via POST or PUT requests), Flask provides utilities to parse it.

Example: Adding a User

```
@app.route('/add-user', methods=['POST'])
def add_user():
    data = request.get_json()    # Parse JSON from request body

    name = data.get('name')
    email = data.get('email')

    if not name or not email:
        return jsonify({"error": "Name and email are required"}), 400

    return jsonify({
        "message": "User added successfully",
        "user": {
            "name": name,
            "email": email
        }
    }), 201
```

Explanation:

- `request.get_json()` extracts JSON payload from the request.
- Input is validated before proceeding.
- A success response is sent back with a `201 Created` status.

Test with **cURL**:

```
curl -X POST http://127.0.0.1:5000/add-user \
-H "Content-Type: application/json" \
-d '{"name": "Vinod", "email": "vinod@vinod.co}"'
```

Returning JSON Responses

Flask makes sending JSON back to the client easy using the `jsonify` helper.

Example: Fetching All Users

```
@app.route('/get-users', methods=['GET'])
def get_users():
    users = [
        {"id": 1, "name": "Vinod", "email": "vinod@vinod.co"},
        {"id": 2, "name": "Shyam", "email": "shyam@xmpl.com"}
    ]
    return jsonify(users)
```

Here, `jsonify`:

- Converts Python dictionaries/lists into JSON.
- Automatically sets the correct `Content-Type: application/json` header.

Error Handling with JSON

A well-designed API should send consistent error messages in JSON format. Instead of raw HTML error pages, JSON responses are machine-readable and easier to consume.

Example 1: Validation Error

```
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    email = data.get('email')

    if not email or "@" not in email:
        return jsonify({"error": "Invalid email address"}), 400

    return jsonify({"message": "Registration successful"}), 201
```

Example 2: Not Found Error

```
@app.route('/get-user/<int:user_id>', methods=['GET'])
def get_user(user_id):
    users = {
        1: {"name": "Vinod", "email": "vinod@vinod.co"},
        2: {"name": "Shyam", "email": "shyam@xmpl.com"}
    }

    user = users.get(user_id)
    if not user:
        return jsonify({"error": "User not found"}), 404

    return jsonify(user)
```

Tips & Best Practices

1. **Validate Input:** Never trust client data. Validate JSON payloads using libraries like [Marshmallow](#) or [pydantic](#).
2. **Consistent Response Format:** Define a standard response structure for success and error cases.
3. **Use Status Codes:** Return appropriate HTTP status codes (200, 201, 400, 404, etc.).
4. **Content-Type Matters:** Ensure clients set `Content-Type: application/json` in request headers for JSON requests.

Running the App

Save the code in a file, say `app.py`, and run:

```
flask run
```

Test with **cURL**:

```
curl -X GET http://127.0.0.1:5000/get-users
```

Response:

```
[
  { "id": 1, "name": "Vinod", "email": "vinod@vinod.co" },
  { "id": 2, "name": "Shyam", "email": "shyam@xmpl.com" }
]
```

Conclusion

Handling JSON in Flask is straightforward with `request.get_json()` for incoming data and `jsonify()` for outgoing responses. By adding input validation, proper error handling, and consistent response structures, you can build robust and reliable APIs that integrate seamlessly with modern applications.

Request Validation & Error Handling in Flask Application

When building web applications, one of the most important aspects is **ensuring data integrity** and **handling errors gracefully**. A Flask application without proper request validation may end

up with unexpected inputs, while an app without robust error handling might confuse users with cryptic error messages.

In this post, we'll explore how to perform request validation and error handling in a **Flask application** using practical examples.

Why Request Validation Matters

Request validation ensures that the data coming into your API is clean, accurate, and follows the rules you've defined. Without validation, you risk:

- **Bad data in your database** (e.g., invalid emails, missing required fields).
- **Application crashes** due to unexpected input types.
- **Security vulnerabilities**, since malicious users can send invalid or malicious payloads.

Setting up a Simple Flask App

Let's create a small API where users can submit their details. For demonstration, we'll use two users:

- **Vinod** → `vinod@vinod.co`
- **Shyam** → `shyam@xmpl.com`

Flask Project Structure

```
flask-validation/  
|  
├─ app.py  
└─ requirements.txt
```

Basic `app.py`

```
from flask import Flask, request, jsonify  
  
app = Flask(__name__)  
  
# In-memory data store  
users = [  
    {"name": "Vinod", "email": "vinod@vinod.co"},  
    {"name": "Shyam", "email": "shyam@xmpl.com"}  
]
```

Adding Request Validation

Let's say we want to add new users. The client must send `name` and `email`. We'll validate the request before inserting into our data store.

```
from email_validator import validate_email, EmailNotValidError

@app.route('/users', methods=['POST'])
def add_user():
    data = request.get_json()

    # Check if required fields are present
    if not data or 'name' not in data or 'email' not in data:
        return jsonify({"error": "Name and email are required"}), 400

    # Validate email format
    try:
        validate_email(data['email'])
    except EmailNotValidError:
        return jsonify({"error": "Invalid email format"}), 400

    # Save user
    new_user = {"name": data['name'], "email": data['email']}
    users.append(new_user)

    return jsonify({"message": "User added successfully", "user":
new_user}), 201
```

✓ Here, we used the **email-validator** library to check if the email is valid. If the validation fails, the API returns a **400 Bad Request** error with a clear message.

Error Handling in Flask

Errors are inevitable—what matters is how we handle them. Flask allows us to define **error handlers** for specific exceptions or HTTP error codes.

Example: Handling “Not Found” Error

If a client tries to fetch a user that doesn't exist, we should return a **404 Not Found** error instead of a blank response.

```
@app.route('/users/<string:name>', methods=['GET'])
def get_user(name):
    for user in users:
        if user['name'].lower() == name.lower():
            return jsonify(user)

    # User not found
    return jsonify({"error": f"User '{name}' not found"}), 404
```

If someone requests `/users/Ravi`, they'll get:

```
{
  "error": "User 'Ravi' not found"
}
```

Global Error Handlers

Instead of handling errors in every route, you can define global error handlers:

```
@app.errorhandler(400)
def bad_request(error):
    return jsonify({"error": "Bad Request", "message": str(error)}),
    400

@app.errorhandler(404)
def not_found(error):
    return jsonify({"error": "Resource not found"}), 404

@app.errorhandler(500)
def internal_error(error):
    return jsonify({"error": "Something went wrong. Please try again
    later."}), 500
```

With this setup, any `400`, `404`, or `500` errors will automatically return JSON responses—making your API consistent and user-friendly.

Testing the API

Add a new user (valid request)

```
curl -X POST http://localhost:5000/users \
-H "Content-Type: application/json" \
-d '{"name": "Ravi", "email": "ravi@example.com}"'
```

✓ Response:

```
{
  "message": "User added successfully",
  "user": {
    "name": "Ravi",
    "email": "ravi@example.com"
  }
}
```

Add a new user (invalid email)

```
curl -X POST http://localhost:5000/users \
-H "Content-Type: application/json" \
-d '{"name": "Ravi", "email": "invalid-email"}'
```

✗ Response:

```
{
  "error": "Invalid email format"
}
```

Get non-existing user

```
curl http://localhost:5000/users/Ravi
```

✗ Response:

```
{
  "error": "User 'Ravi' not found"
}
```

Conclusion

By combining **request validation** and **error handling**, you can make your Flask applications **more reliable, secure, and user-friendly**.

- Validation ensures only clean and expected data enters your system.

- Error handling ensures that unexpected issues don't break your app but instead return meaningful messages.

These small practices make a big difference in creating production-ready Flask applications.

Global error handling example

Flask App (with `abort()` for errors)

```
from flask import Flask, request, jsonify, abort
from email_validator import validate_email, EmailNotValidError

app = Flask(__name__)

# In-memory users list
users = [
    {"name": "Vinod", "email": "vinod@vinod.co"},
    {"name": "Shyam", "email": "shyam@xmpl.com"}
]

# -----
# Routes
# -----

@app.route("/users", methods=["POST"])
def add_user():
    """Add a new user after validation"""
    data = request.get_json()

    if not data or "name" not in data or "email" not in data:
        abort(400, description="Name and email are required")

    try:
        validate_email(data["email"])
    except EmailNotValidError:
        abort(400, description="Invalid email format")

    new_user = {"name": data["name"], "email": data["email"]}
    users.append(new_user)
    return jsonify({"message": "User added", "user": new_user}), 201

@app.route("/users/<string:name>", methods=["GET"])
def get_user(name):
    """Fetch a user by name"""
    for user in users:
        if user["name"].lower() == name.lower():
            return jsonify(user)
```

```

    abort(404, description=f"User '{name}' not found")

@app.route("/crash", methods=["GET"])
def crash_app():
    """Force a crash to test 500 handler"""
    abort(500, description="Forced server crash for testing")

# -----
# Global Error Handlers
# -----

@app.errorhandler(400)
def handle_bad_request(error):
    return jsonify({
        "status": 400,
        "error": "Bad Request",
        "message": error.description
    }), 400

@app.errorhandler(404)
def handle_not_found(error):
    return jsonify({
        "status": 404,
        "error": "Resource Not Found",
        "message": error.description
    }), 404

@app.errorhandler(500)
def handle_internal_error(error):
    return jsonify({
        "status": 500,
        "error": "Internal Server Error",
        "message": error.description or "Something went wrong on the
server."
    }), 500

# -----
# Run Flask app
# -----
if __name__ == "__main__":
    app.run(debug=True)

```

Behavior

1. POST with missing fields

```
curl -X POST http://127.0.0.1:5000/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Ravi"}'
```

Response:

```
{  
  "status": 400,  
  "error": "Bad Request",  
  "message": "Name and email are required"  
}
```

2. GET non-existing user

```
curl http://127.0.0.1:5000/users/Ravi
```

Response:

```
{  
  "status": 404,  
  "error": "Resource Not Found",  
  "message": "User 'Ravi' not found"  
}
```

3. Crash the server

```
curl http://127.0.0.1:5000/crash
```

Response:

```
{  
  "status": 500,  
  "error": "Internal Server Error",  
  "message": "Forced server crash for testing"  
}
```

This way, **all errors go through the global error handlers**, ensuring a consistent JSON response schema across your API 🚀.

Building REST APIs with Flask-RESTful

When working with **Flask**, you can quickly build web applications, but if your goal is to expose a **REST API**, the **Flask-RESTful** extension makes the process much cleaner and more structured. It provides tools for defining resources, handling requests, and standardizing responses with minimal boilerplate.

In this post, we'll walk through building a simple REST API using **Flask-RESTful**. We'll manage a list of users and learn how to handle errors gracefully.

Why Flask-RESTful?

Out of the box, Flask lets you define routes and handle requests:

```
@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)
```

This works, but as your API grows, the routes become messy and hard to maintain. Flask-RESTful organizes things into **resources**, which map directly to **REST endpoints**.

Setting Up

Install Flask-RESTful with pip:

```
pip install flask-restful
```

Then, import it into your project:

```
from flask import Flask
from flask_restful import Api, Resource, reqparse
```

Creating a Simple Resource

We'll build a **User Resource**. A resource corresponds to a single endpoint and defines HTTP methods like **GET**, **POST**, **PUT**, and **DELETE**.

Let's start with some in-memory data:

```
users = [
    {"name": "Vinod", "email": "vinod@vinod.co"},
    {"name": "Shyam", "email": "shyam@xmpl.com"}
]
```


Now, define a resource:

```
class UserList(Resource):  
    def get(self):  
        return {"users": users}, 200
```

And connect it to an endpoint:

```
api.add_resource(UserList, "/users")
```

When you `GET /users`, you'll receive the full list of users.

Adding Request Parsing

We want to allow adding new users via a `POST` request. Flask-RESTful provides `reqparse` to validate inputs.

```
parser = reqparse.RequestParser()  
parser.add_argument("name", type=str, required=True, help="Name  
cannot be blank!")  
parser.add_argument("email", type=str, required=True, help="Email  
cannot be blank!")
```

Inside the resource:

```
class UserList(Resource):  
    def get(self):  
        return {"users": users}, 200  
  
    def post(self):  
        data = parser.parse_args()  
        users.append({"name": data["name"], "email": data["email"]})  
        return {"message": "User added successfully!"}, 201
```

Handling Errors Gracefully

APIs should provide **clear error responses**. Flask-RESTful makes this easy by raising exceptions with messages.

Example 1: Validation Error

If you call `POST /users` without the required fields, you'll get:

```
{
  "message": {
    "name": "Name cannot be blank!",
    "email": "Email cannot be blank!"
  }
}
```

Example 2: Not Found Error

Let's create a resource to get a user by name:

```
class User(Resource):
    def get(self, name):
        for user in users:
            if user["name"].lower() == name.lower():
                return user, 200
        return {"error": "User not found"}, 404
```

So, if you request `GET /users/Ravi`, the API will return:

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

Complete Code

Here's the full example you can run directly:

```
from flask import Flask
from flask_restful import Api, Resource, reqparse

app = Flask(__name__)
api = Api(app)

users = [
    {"name": "Vinod", "email": "vinod@vinod.co"},
    {"name": "Shyam", "email": "shyam@xmpl.com"}
]

parser = reqparse.RequestParser()
parser.add_argument("name", type=str, required=True, help="Name cannot be blank!")
parser.add_argument("email", type=str, required=True, help="Email cannot be blank!")

class UserList(Resource):
    def get(self):
        return {"users": users}, 200

    def post(self):
        data = parser.parse_args()
        users.append({"name": data["name"], "email": data["email"]})
        return {"message": "User added successfully!"}, 201

class User(Resource):
    def get(self, name):
        for user in users:
            if user["name"].lower() == name.lower():
                return user, 200
        return {"error": "User not found"}, 404

api.add_resource(UserList, "/users")
api.add_resource(User, "/users/<string:name>")

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

Conclusion

With just a few lines of code, Flask-RESTful helps you structure your APIs in a **clean, scalable, and maintainable** way. It takes care of common concerns like parsing requests, error handling, and mapping HTTP methods to class methods.

This is just the beginning—you can extend this project with authentication, database integration, and pagination. But even at this stage, you have a well-structured API ready for

real-world use.

API Versioning Strategies in Flask

When building APIs, one of the **biggest challenges** is handling changes over time without breaking existing clients. Imagine you've deployed your API, and mobile apps or other services are consuming it. If you suddenly change the structure of responses or remove fields, those clients may fail.

That's where **API versioning** comes in. Versioning allows you to evolve your API without breaking older clients. In this post, we'll explore **API versioning strategies in Flask**, with practical examples.

Why Versioning Matters

Consider a simple users API that returns:

```
{
  "name": "Vinod",
  "email": "vinod@vinod.co"
}
```

Later, you decide to also include a `phone` field. If a client app wasn't built to handle `phone`, it may ignore it — but what if you rename `email` to `contactEmail`? That could break existing apps.

By versioning your API, you can support both **old** and **new** clients at the same time.

Common API Versioning Strategies

1. URI Path Versioning

The simplest and most common approach. Add the version number directly into the URL.

```
@app.route('/api/v1/users')
def get_users_v1():
    return [{"name": "Vinod", "email": "vinod@vinod.co"}]

@app.route('/api/v2/users')
def get_users_v2():
    return [{"name": "Vinod", "email": "vinod@vinod.co", "phone":
"123-456-7890"}]
```

Clients choose which version they want by calling `/api/v1/...` or `/api/v2/...`

✅ Easy to implement ✅ Clear separation ❌ URLs can get cluttered

2. Query Parameter Versioning

Clients specify the version using a query string.

```
@app.route('/api/users')
def get_users():
    version = request.args.get('version', '1')
    if version == '2':
        return [{"name": "Vinod", "email": "vinod@vinod.co", "phone":
"123-456-7890"}]
    return [{"name": "Vinod", "email": "vinod@vinod.co"}]
```

Clients call:

- `/api/users?version=1`
- `/api/users?version=2`

✅ Simple to implement ❌ Easy to forget adding version parameter ❌ Can clutter query strings

3. Header-Based Versioning

Clients specify the version using a custom HTTP header.

```
@app.route('/api/users')
def get_users():
    version = request.headers.get("X-API-Version", "1")
    if version == "2":
        return [{"name": "Vinod", "email": "vinod@vinod.co", "phone":
"123-456-7890"}]
    return [{"name": "Vinod", "email": "vinod@vinod.co"}]
```

Client request example:

```
curl -H "X-API-Version: 2" http://localhost:5000/api/users
```

✅ Keeps URLs clean ✅ Easy to migrate clients silently ❌ Slightly harder to test manually

4. Accept Header (Content Negotiation)

Clients specify version in the `Accept` header.

```
@app.route('/api/users')
def get_users():
    accept = request.headers.get("Accept", "application/json")
    if "vnd.myapi.v2+json" in accept:
        return [{"name": "Vinod", "email": "vinod@vinod.co", "phone":
"123-456-7890"}]
    return [{"name": "Vinod", "email": "vinod@vinod.co"}]
```

Client request example:

```
curl -H "Accept: application/vnd.myapi.v2+json"
http://localhost:5000/api/users
```

✅ Industry-standard approach ✅ Works well with API gateways ❌ More complex for beginners

Error Handling in Versioning

It's important to handle unsupported versions gracefully. For example, if a client asks for version `3` that doesn't exist:

```
if version not in ["1", "2"]:
    return {"error": "API version not supported"}, 400
```

Clients should get a **clear error message** instead of a broken response.

Complete Example: URI Path Versioning

Here's a runnable Flask app demonstrating **URI path versioning**:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/v1/users')
def get_users_v1():
    return jsonify([
        {"name": "Vinod", "email": "vinod@vinod.co"},
        {"name": "Shyam", "email": "shyam@xmpl.com"}
    ])

@app.route('/api/v2/users')
def get_users_v2():
    return jsonify([
        {"name": "Vinod", "email": "vinod@vinod.co", "phone": "123-456-7890"},
        {"name": "Shyam", "email": "shyam@xmpl.com", "phone": "987-654-3210"}
    ])

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

Conclusion

API versioning is crucial for maintaining **backward compatibility** while evolving your API.

- **Path versioning** is the simplest.
- **Header-based and Accept-header strategies** are more professional and align with modern API design.
- **Query parameter versioning** is quick but less recommended for large-scale systems.

When you design APIs in Flask (or any framework), plan your **versioning strategy upfront**. This avoids painful breaking changes later and ensures a smooth experience for your clients.

Statelessness and Caching Concepts in REST APIs

When designing REST APIs, two fundamental principles that directly impact **performance, scalability, and reliability** are **statelessness** and **caching**. These concepts ensure that your API can handle high traffic, scale horizontally, and provide fast responses to clients.

In this post, we'll break down what **statelessness** means, why it's important, and how **caching** complements it to improve performance.

What Does Statelessness Mean?

In REST architecture, **statelessness** means that the server does **not store any information about the client's state** between requests. Each request from a client to a server must contain all the information needed to understand and process it.

Example

Imagine an API that manages users. If you request user details:

```
GET /api/users/Vinod
```

The server returns:

```
{
  "name": "Vinod",
  "email": "vinod@vinod.co"
}
```

The server does not remember that you just retrieved Vinod's profile. If you want to fetch Shyam's profile, you make another request:

```
GET /api/users/Shyam
```

Each request is **independent**, and the server does not rely on previous interactions.

Benefits of Statelessness

- **Scalability** – Any request can be served by any server in a cluster since no session data is stored on the server.
- **Reliability** – Failures on one server do not affect ongoing sessions, as there are no sessions to begin with.
- **Simplicity** – The server's job is only to process requests, not manage user sessions.

The Role of Caching in REST APIs

While statelessness ensures scalability, **caching** ensures **performance efficiency**. REST encourages the use of **HTTP caching** to avoid recomputing or refetching the same data repeatedly.

Example

Suppose you request user details for Vinod multiple times:

```
GET /api/users/Vinod
```

Without caching, the server processes each request anew. With caching, the response can be stored and reused until it expires.

Caching Headers

REST APIs typically use **HTTP headers** to control caching.

1. **Cache-Control**

Defines caching policies.

```
Cache-Control: max-age=3600
```

This means the response can be cached for **1 hour**.

2. **ETag** (Entity Tag)

Allows conditional requests. The server assigns a unique identifier (hash) to a response. If the client already has this data cached, it sends the ETag in a header:

```
If-None-Match: "abc123"
```

If the resource hasn't changed, the server responds with:

```
304 Not Modified
```

— saving bandwidth and speeding up responses.

3. **Expires**

Sets a fixed expiration time for cached data.

```
Expires: Wed, 23 Aug 2025 12:00:00 GMT
```

Combining Statelessness and Caching

- Statelessness ensures every request is **self-contained**.
- Caching ensures frequently requested resources are served **faster**.

Together, they provide the backbone of efficient RESTful systems.

Practical Example in Flask

Here's a simple Flask API implementing **statelessness and caching**:

```
from flask import Flask, jsonify, make_response

app = Flask(__name__)

users = {
    "Vinod": {"name": "Vinod", "email": "vinod@vinod.co"},
    "Shyam": {"name": "Shyam", "email": "shyam@xmpl.com"}
}

@app.route("/api/users/<string:name>")
def get_user(name):
    user = users.get(name)
    if not user:
        return {"error": "User not found"}, 404

    response = make_response(jsonify(user), 200)
    response.headers["Cache-Control"] = "public, max-age=60" # cache
    for 1 min
    response.headers["ETag"] = "user-" + name.lower()
    return response

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

How this works:

- Every request is **stateless** — no session is stored.
- Responses include **Cache-Control** and **ETag** headers, making them cache-friendly.

Conclusion

- **Statelessness** makes REST APIs scalable, reliable, and easy to maintain.
- **Caching** reduces latency and server load by reusing responses.
- Together, they help build high-performance APIs that scale with demand.

When designing APIs in Flask (or any other framework), always ensure **stateless design** and use **caching headers** effectively.

Flask-SQLAlchemy Integration

Most modern applications need a backend that can manage **structured data** efficiently — products, users, orders, categories, and so on. Flask, with its minimalistic approach, makes a great choice for building such applications. Combined with **SQLAlchemy** (ORM for Python), you can manage database interactions cleanly without writing raw SQL all the time.

In this step-by-step guide, we'll build a **Flask + SQLAlchemy application** that manages **Products and Categories** with support for:

- Models and relationships (One-to-Many, Many-to-Many)
- Migration management with **Flask-Migrate**
- Query optimization
- Role-Based Access Control (RBAC) strategies

We'll use **products** as examples, so the data feels familiar and practical.

◆ Step 1: Setup the Environment

Install required packages:

```
pip install flask flask_sqlalchemy flask_migrate flask-bcrypt flask-jwt-extended
```

Project structure:

```
flask_sqlalchemy_products/  
|— app.py  
|— models.py  
|— routes.py  
|— config.py  
|— migrations/  
|— database.db
```

◆ Step 2: Configure Flask and Database

config.py

```
import os

BASE_DIR = os.path.abspath(os.path.dirname(__file__))

class Config:
    SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(BASE_DIR,
    'database.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = "supersecretkey"
```

app.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from config import Config

db = SQLAlchemy()
migrate = Migrate()

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    db.init_app(app)
    migrate.init_app(app, db)

    from routes import main
    app.register_blueprint(main)

    return app

app = create_app()

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

◆ Step 3: Define Models & Relationships

We'll have:

- **Category** (e.g., Electronics, Groceries)
- **Product** (belongs to one category, can have multiple tags)
- **Tag** (Many-to-Many relationship with Products)
- **User** (with roles for RBAC: admin, manager, customer)

models.py

```

from app import db
from flask_bcrypt import Bcrypt

bcrypt = Bcrypt()

# Association table for Many-to-Many
product_tags = db.Table('product_tags',
    db.Column('product_id', db.Integer, db.ForeignKey('product.id'),
primary_key=True),
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'),
primary_key=True)
)

class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True, nullable=False)
    products = db.relationship('Product', backref='category',
lazy=True)

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    price = db.Column(db.Float, nullable=False)
    stock = db.Column(db.Integer, default=0)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'),
nullable=False)
    tags = db.relationship('Tag', secondary=product_tags,
lazy='subquery',
backref=db.backref('products', lazy=True))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    password_hash = db.Column(db.String(200), nullable=False)
    role = db.Column(db.String(50), nullable=False,
default="customer") # admin, manager, customer

    def set_password(self, password):
        self.password_hash =
bcrypt.generate_password_hash(password).decode('utf-8')

    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash,
password)

```

◆ Step 4: Database Migrations with Flask-Migrate

Instead of manually creating tables, we'll use **Flask-Migrate**.

```
flask db init
flask db migrate -m "Initial migration"
flask db upgrade
```

This will generate versioned migrations inside the **migrations/** folder.

Now you can evolve your database schema safely over time.

◆ Step 5: Add Routes for CRUD Operations

routes.py

```
from flask import Blueprint, request, jsonify
from app import db
from models import Category, Product, Tag, User

main = Blueprint('main', __name__)

# Create a category
@main.route('/category', methods=['POST'])
def create_category():
    data = request.json
    new_category = Category(name=data['name'])
    db.session.add(new_category)
    db.session.commit()
    return jsonify({"message": "Category created"})

# Get categories with products (optimized query)
@main.route('/categories', methods=['GET'])
def get_categories():
    categories =
Category.query.options(db.joinedload(Category.products)).all()
    return jsonify([
        {
            "id": c.id,
            "name": c.name,
            "products": [{"id": p.id, "name": p.name, "price":
p.price} for p in c.products]
        }
        for c in categories
    ])

# Add a product with category & tags
```

```
@main.route('/product', methods=['POST'])
def create_product():
    data = request.json
    category = Category.query.get(data['category_id'])
    if not category:
        return jsonify({"error": "Category not found"}), 404

    product = Product(
        name=data['name'],
        price=data['price'],
        stock=data.get('stock', 0),
        category=category
    )

    if 'tags' in data:
        for tag_name in data['tags']:
            tag = Tag.query.filter_by(name=tag_name).first()
            if not tag:
                tag = Tag(name=tag_name)
                db.session.add(tag)
            product.tags.append(tag)

    db.session.add(product)
    db.session.commit()
    return jsonify({"message": "Product created successfully"})
```

◆ Step 6: Insert sample Data

```
curl -X POST http://127.0.0.1:5000/category \
-H "Content-Type: application/json" \
-d '{"name": "Electronics"}'

curl -X POST http://127.0.0.1:5000/product \
-H "Content-Type: application/json" \
-d '{"name": "Mi LED TV", "price": 32999, "stock": 25, "category_id": 1, "tags": ["TV", "Smart"]}'
```

We now have **Electronics** → **Mi LED TV (tags: TV, Smart)**. You can add more: **"Tata Salt"**, **"Amul Butter"**, **"Samsung Galaxy S23"** etc.

◆ Step 7: Query Optimization

Instead of multiple queries (N+1 problem), use:


```
from sqlalchemy.orm import joinedload

categories =
Category.query.options(joinedload(Category.products)).all()
```

This fetches **categories and products in one go**, improving performance.

Other strategies:

- Use `.with_entities()` to fetch only required columns.
- Use `.filter_by()` and `.filter()` with indexes.
- Apply pagination (`.limit().offset()`).

◆ Step 8: Role-Based Access Control (RBAC)

We'll keep RBAC simple:

- **Admin:** Can create categories/products, delete anything
- **Manager:** Can update stock/prices
- **Customer:** Can only view products

Example check:

```
def check_role(user, required_role):
    roles = {"admin": 3, "manager": 2, "customer": 1}
    return roles[user.role] >= roles[required_role]

@main.route('/secure-add-product', methods=['POST'])
def secure_add_product():
    data = request.json
    user = User.query.filter_by(username=data['username']).first()
    if not user or not user.check_password(data['password']):
        return jsonify({"error": "Unauthorized"}), 401

    if not check_role(user, "admin"):
        return jsonify({"error": "Permission denied"}), 403

    # proceed with product creation...
    return jsonify({"message": "Product added securely!"})
```

This way, **only admins** can add products, while managers and customers have restricted privileges.

◆ Step 9: Run the App

```
python app.py
```

Open: <http://127.0.0.1:5000/categories>

◆ Conclusion

In this tutorial, we've covered:

✅ Setting up Flask with SQLAlchemy ✅ Defining models with **1-to-Many & Many-to-Many relationships** ✅ Managing schema changes with **Flask-Migrate** ✅ Optimizing queries for better performance ✅ Implementing a simple **RBAC strategy**

This provides a **production-ready foundation** for an e-commerce or inventory management system with **Products & Categories**.

👉 Next Steps:

- Add **JWT-based authentication** using `flask-jwt-extended`
- Add **pagination & sorting** for large product lists
- Add **unit tests** for APIs

Authentication & Authorization in REST APIs

Modern applications rarely live in isolation — they usually expose REST APIs consumed by web apps, mobile apps, or third-party systems. In such setups, **authentication (who are you?)** and **authorization (what can you do?)** are critical to ensure secure access.

In this tutorial, we'll explore the **three most popular mechanisms** used in industry:

- JSON Web Tokens (**JWT**)
- OAuth2
- Single Sign-On (**SSO**) with providers like **Google** or **Okta**

We'll also build practical demos with Flask so you can implement these in your projects.

Part 1: Authentication vs Authorization

Before diving into the details, let's clear the difference:

- **Authentication:** Verifying the identity of the user (e.g., logging in with username/password).
- **Authorization:** Determining what the authenticated user can access (e.g., admin can add products, customer can only view products).

Think of it like this:

- Entering a cinema hall: showing your ticket = **authentication**.
- Which seat you're allowed to sit in = **authorization**.

Part 2: JWT Authentication

JWT (JSON Web Token) is one of the most common ways to secure REST APIs.

A JWT consists of 3 parts:

```
header.payload.signature
```

Example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6I.. (header)
eyJ1c2VyX2lkIjoxLCJyb2xlIjoieWRtaW4ifQ.. (payload)
jL2ZP7jAn5ZcUPeJ5qNcl7Up.. (signature)
```

Demo: JWT Authentication in Flask

Install dependencies:

```
pip install flask flask_sqlalchemy flask-jwt-extended flask-bcrypt
```

app.py

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_jwt_extended import JWTManager, create_access_token,
jwt_required, get_jwt_identity

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///users.db"
app.config["SECRET_KEY"] = "supersecret"
app.config["JWT_SECRET_KEY"] = "jwtsecret"

db = SQLAlchemy(app)
bcrypt = Bcrypt(app)
jwt = JWTManager(app)

class User(db.Model):
    id = db.Column(db.Integer, primary key=True)
```

```

username = db.Column(db.String(80), unique=True, nullable=False)
password_hash = db.Column(db.String(200), nullable=False)
role = db.Column(db.String(50), default="customer")

def set_password(self, password):
    self.password_hash =
bcrypt.generate_password_hash(password).decode('utf-8')

def check_password(self, password):
    return bcrypt.check_password_hash(self.password_hash,
password)

@app.route("/register", methods=["POST"])
def register():
    data = request.json
    user = User(username=data["username"])
    user.set_password(data["password"])
    db.session.add(user)
    db.session.commit()
    return jsonify({"message": "User registered successfully!"})

@app.route("/login", methods=["POST"])
def login():
    data = request.json
    user = User.query.filter_by(username=data["username"]).first()
    if user and user.check_password(data["password"]):
        token = create_access_token(identity=user.username,
additional_claims={"role": user.role})
        return jsonify({"access_token": token})
    return jsonify({"error": "Invalid credentials"}), 401

@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    identity = get_jwt_identity()
    return jsonify({"message": f"Welcome {identity['username']}!",
"role": identity['role']})

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

```

Testing JWT

```
# Register
curl -X POST http://127.0.0.1:5000/register \
  -H "Content-Type: application/json" \
  -d '{"username": "vinod", "password": "mypassword"}'

# Login (get token)
curl -X POST http://127.0.0.1:5000/login \
  -H "Content-Type: application/json" \
  -d '{"username": "vinod", "password": "mypassword"}'

# Use token in protected route
curl http://127.0.0.1:5000/protected \
  -H "Authorization: Bearer <your_token_here>"
```

JWT is great for **stateless APIs** where the token is stored on the client (mobile app, frontend app).

Part 3: OAuth2

JWT works well for internal systems, but when external apps need to access resources on behalf of a user, **OAuth2** is the industry standard.

- **Resource Owner** → You (the user)
- **Client** → The app requesting access (say Zomato wants your Google Contacts)
- **Authorization Server** → Identity provider (Google, Facebook, Okta)
- **Resource Server** → API hosting your data

OAuth2 Example with Google

We'll use `flask-dance` (simplifies OAuth2 in Flask).

Install:

```
pip install flask-dance
```

google_auth.py

```
from flask import Flask, redirect, url_for
from flask_dance.contrib.google import make_google_blueprint, google

app = Flask(__name__)
app.secret_key = "supersecret"
app.config["GOOGLE_OAUTH_CLIENT_ID"] = "<your_client_id>"
app.config["GOOGLE_OAUTH_CLIENT_SECRET"] = "<your_client_secret>"

google_bp = make_google_blueprint(scope=["profile", "email"])
app.register_blueprint(google_bp, url_prefix="/login")

@app.route("/")
def index():
    if not google.authorized:
        return redirect(url_for("google.login"))
    resp = google.get("/oauth2/v2/userinfo")
    profile = resp.json()
    return f"Hello {profile['email']}! Welcome via Google OAuth2."
```

Now your users can **log in with Google**.

Part 4: Single Sign-On (SSO)

SSO allows users to log in once and gain access to multiple applications.

Example: If your company uses **Okta** or **Google Workspace**, you can sign in once and access Gmail, Google Drive, Calendar, etc.

SSO is usually implemented using:

- **SAML (XML-based, older standard)**
- **OAuth2 + OpenID Connect (modern standard)**

Demo: Flask + Okta SSO

1. Sign up for a free **Okta developer account** at <https://developer.okta.com>.
2. Create a new **OIDC app** in Okta dashboard.
3. Configure redirect URI: `http://localhost:5000/authorization-code/callback`.
4. Install dependencies:

```
pip install flask flask_oidc
```

okta_demo.py

```
from flask import Flask, jsonify
from flask_oidc import OpenIDConnect

app = Flask(__name__)
app.config.update({
    "SECRET_KEY": "supersecret",
    "OIDC_CLIENT_SECRETS": "client_secrets.json",
    "OIDC_RESOURCE_SERVER_ONLY": True,
    "OIDC_INTROSPECTION_AUTH_METHOD": "client_secret_post",
})

oidc = OpenIDConnect(app)

@app.route("/profile")
@oidc.require_login
def profile():
    user_info = oidc.user_getinfo(["email", "sub"])
    return jsonify(user_info)

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

Here, `client_secrets.json` comes from your Okta application settings.

JWT vs OAuth2 vs SSO

Feature	JWT	OAuth2	SSO (Okta/Google)
Use Case	API authentication	Third-party app access	One login for multiple apps
Token Type	Self-contained JWT	Access & refresh tokens	SAML / OIDC tokens
Storage	Client (browser/app)	Authorization server	Central identity provider
Example	Your internal API	Login with Google on Zomato	Okta login for enterprise

Conclusion

In this guide, we:

- ✔ Implemented **JWT authentication** for REST APIs
- ✔ Understood **OAuth2 flow** and integrated Google login
- ✔ Explored **SSO with Okta**
- ✔ Compared JWT, OAuth2, and SSO with

real-world use cases

These three mechanisms cover **90% of authentication/authorization use cases** in modern systems.

👉 Next steps you can try:

- Add **role-based authorization** to JWT routes.
- Extend **OAuth2 integration** to access Google Drive or Calendar.
- Deploy **Okta-based SSO** for your corporate apps.

API Rate Limiting & Throttling in Flask with Flask-Limiter

APIs are powerful, but without **rate limiting** they can easily be abused. Imagine a malicious user (or even a buggy client) sending **thousands of requests per second** — your server could slow down, crash, or run up unexpected costs.

This is where **API rate limiting & throttling** comes in. In this tutorial, we'll explore:

- What API rate limiting is and why it matters
- Difference between **rate limiting** and **throttling**
- Setting up **Flask-Limiter** in a Flask REST API
- Applying limits globally, per-route, and per-user
- Using **custom keys** (like API keys, JWT user IDs, or IP addresses)
- Handling exceeded limits gracefully
- A practical **e-commerce product API example**

Part 1: What is Rate Limiting?

- **Rate limiting** → Restricts how many requests a client can make in a given timeframe (e.g., 100 requests per minute).
- **Throttling** → Slows down or blocks requests once the rate limit is reached.

Example policies:

- Anonymous users: **10 requests/minute**
- Authenticated users: **100 requests/minute**
- Admins: **unlimited**

Part 2: Install Dependencies

```
pip install flask flask-restful flask-limiter
```


Part 3: Project Setup

app.py

```
from flask import Flask, request, jsonify
from flask_restful import Api, Resource
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)
api = Api(app)

# Initialize Limiter
limiter = Limiter(
    get_remote_address, # Default: limit by client IP
    app=app,
    default_limits=["100 per hour"] # Global default
)
```

Part 4: Example API (Products)

We'll build a simple **e-commerce products API**.

```
products = [
    {"id": 1, "name": "Amul Butter", "price": 50},
    {"id": 2, "name": "Tata Tea", "price": 200},
    {"id": 3, "name": "Parle-G Biscuits", "price": 10},
]

@limiter.limit("5 per minute")
class ProductList(Resource):

    def get(self):
        return jsonify(products)

@limiter.limit("10 per minute")
class ProductDetail(Resource):

    def get(self, product_id):
        product = next((p for p in products if p["id"] ==
product_id), None)
        if product:
            return jsonify(product)
        return {"error": "Product not found"}, 404
```

Part 5: Register Endpoints

```
api.add_resource(ProductList, "/products")
api.add_resource(ProductDetail, "/products/<int:product_id>")
```

Run the app:

```
python app.py
```

Part 6: Testing the Rate Limits

Fetch all products (allowed: 5 requests/minute)

```
curl http://127.0.0.1:5000/products
```

If you exceed 5 requests per minute:

```
{
  "error": "429 Too Many Requests: The rate limit is exceeded."
}
```

Part 7: Different Limits for Different Users

Instead of limiting **per IP**, you can rate limit **per user** using API keys or JWT.

Example with **API key from headers**:

```
def get_api_key():
    return request.headers.get("X-API-KEY") or get_remote_address

limiter = Limiter(
    key_func=get_api_key,
    app=app,
    default_limits=["50 per hour"]
)
```

Now:

- Each API key has its own quota

- Without a key → falls back to IP address

Part 8: Custom Limits for Roles (User vs Admin)

Let's simulate **role-based throttling**:

```
USER_LIMITS = "10 per minute"
ADMIN_LIMITS = "100 per minute"

def role_based_limit():
    api_key = request.headers.get("X-API-KEY")
    if api_key == "admin123":
        return ADMIN_LIMITS
    return USER_LIMITS

@limiter.limit(role_based_limit)
class RoleBasedProducts(Resource):

    def get(self):
        return jsonify(products)

api.add_resource(RoleBasedProducts, "/rb-products")
```

- Normal user (X-API-KEY=anything else) → 10 requests/min
- Admin (X-API-KEY=admin123) → 100 requests/min

Part 9: Handling Rate Limit Errors Gracefully

By default, Flask-Limiter returns a **429 Too Many Requests** error. We can customize this:

```
@app.errorhandler(429)
def ratelimit_handler(e):
    return jsonify(error="Too many requests. Please try again
later.",
                  limit=str(e.description)), 429
```

Part 10: Advanced Configurations

- **Burst + sustained limits:**

```
@limiter.limit("10 per minute; 200 per day")
def get():
    return jsonify(products)
```

- **Exempt endpoints:**

```
@limiter.exempt
def health_check():
    return {"status": "ok"}
```

- **Shared limits across multiple routes:**

```
shared_limit = limiter.shared_limit("20 per minute",
scope="products")

@app.route("/popular")
@shared_limit
def popular():
    return {"popular": ["Amul Butter", "Tata Tea"]}
```

Conclusion

We've built a **Flask REST API with rate limiting & throttling** using Flask-Limiter.

✅ Global rate limits ✅ Per-route limits ✅ Role-based (user/admin) limits ✅ Custom error handling ✅ Shared limits for groups of endpoints

With these strategies, you can **protect your APIs from abuse, ensure fair usage, and improve stability**.

👉 Next steps:

- Store rate limit data in **Redis/Memcached** for distributed APIs
- Combine with **JWT authentication**
- Apply **dynamic limits** (different tiers: free vs paid users)

File Upload Handling in Flask REST APIs

When building REST APIs, sooner or later you'll need to let users **upload files**—profile photos, documents, receipts, etc. Handling file uploads may look simple, but it requires careful consideration for **security**, **validation**, and **storage**.

In this tutorial, we'll build a small Flask REST API that lets us manage `Customer` records and upload their profile photos.

Our `Customer` model will look like this:

```
Customer { id, name, city, email, phone, photo }
```

1. Project Setup

First, let's create a folder for our project:

```
mkdir flask-file-upload-demo
cd flask-file-upload-demo
mkdir uploads
```

We'll need Flask and SQLAlchemy:

```
python -m venv .venv
source .venv/bin/activate    # macOS/Linux
# .venv\Scripts\activate    # Windows

pip install Flask Flask-SQLAlchemy
```

2. Creating a Flask App

Every Flask project starts with creating the app and setting configurations:

```
from flask import Flask

app = Flask(__name__)

# Configure database and upload settings
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
app.config["UPLOAD_FOLDER"] = "uploads"
app.config["MAX_CONTENT_LENGTH"] = 2 * 1024 * 1024 # 2 MB limit
```

Concept:

- `UPLOAD_FOLDER`: where uploaded files will be stored.
- `MAX_CONTENT_LENGTH`: prevents users from uploading huge files.

3. Defining the Database Model

We want to store customers in a SQLite DB, with their uploaded **photo filename**:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy(app)

class Customer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120), nullable=False)
    city = db.Column(db.String(120))
    email = db.Column(db.String(120), unique=True, nullable=False)
    phone = db.Column(db.String(20), unique=True, nullable=False)
    photo_filename = db.Column(db.String(255))

    def to_dict(self):
        return {
            "id": self.id,
            "name": self.name,
            "city": self.city,
            "email": self.email,
            "phone": self.phone,
            "photo_url": f"/api/customers/{self.id}/photo" if
self.photo_filename else None
        }
```

Concept:

- We don't store the actual image in the DB (bad practice).
- Instead, we store the **filename** and generate a **photo_url** to serve the file.

4. Handling File Uploads Safely

When a file is uploaded in Flask, it's available in `request.files`. But we must:

1. **Validate extension** (e.g., only jpg/png).
2. **Sanitize filename** (avoid malicious paths).
3. **Avoid filename collisions** (use unique IDs).

```

import os
from werkzeug.utils import secure_filename
from uuid import uuid4

ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg", "gif"}

def allowed_file(filename):
    return "." in filename and filename.rsplit(".", 1)[1].lower() in ALLOWED_EXTENSIONS

def save_uploaded_photo(file_storage):
    if file_storage and allowed_file(file_storage.filename):
        original_name = secure_filename(file_storage.filename)
        ext = os.path.splitext(original_name)[1]
        stored_name = f"{uuid4().hex}{ext}"
        file_storage.save(os.path.join(app.config["UPLOAD_FOLDER"], stored_name))
        return stored_name
    return None

```

5. API Endpoints

5.1 Create a Customer with Photo

```

from flask import request, jsonify

@app.route("/api/customers", methods=["POST"])
def create_customer():
    name = request.form.get("name")
    city = request.form.get("city")
    email = request.form.get("email")
    phone = request.form.get("phone")
    photo = request.files.get("photo")

    if not (name and email and phone):
        return jsonify({"error": "name, email, phone are required"}), 400

    filename = save_uploaded_photo(photo)

    customer = Customer(name=name, city=city, email=email, phone=phone, photo_filename=filename)
    db.session.add(customer)
    db.session.commit()

    return jsonify(customer.to_dict()), 201

```

Concept:

- We use `request.form` for text fields and `request.files` for the photo.
- Client must send a **multipart/form-data** request.

5.2 Get Customer List

```
@app.route("/api/customers", methods=["GET"])
def list_customers():
    customers = Customer.query.all()
    return jsonify([c.to_dict() for c in customers])
```

5.3 Serve the Uploaded Photo

```
from flask import send_from_directory, abort

@app.route("/api/customers/<int:id>/photo", methods=["GET"])
def get_customer_photo(id):
    customer = Customer.query.get_or_404(id)
    if not customer.photo_filename:
        abort(404, "No photo found")
    return send_from_directory(app.config["UPLOAD_FOLDER"],
                              customer.photo_filename)
```

Concept:

- `send_from_directory` safely serves files only from the `uploads/` folder.

6. Testing with Sample Data

Now let's test with your **Vinod** example.

Create a customer with photo

```
curl -X POST http://127.0.0.1:5000/api/customers \
-F "name=Vinod" \
-F "city=Bangalore" \
-F "email=vinod@vinod.co" \
-F "phone=9731424784" \
-F "photo=@vinod.jpg"
```

Get customers


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

Response:

```
[
  {
    "id": 1,
    "name": "Vinod",
    "city": "Bangalore",
    "email": "vinod@vinod.co",
    "phone": "9731424784",
    "photo_url": "/api/customers/1/photo"
  }
]
```

Open photo

Go to <http://127.0.0.1:5000/api/customers/1/photo> in your browser.

7. Full Code (for reference)

Here's the complete `app.py` combining all the above:

```
import os
from uuid import uuid4
from flask import Flask, request, jsonify, send_from_directory, url_for, abort
from flask_sqlalchemy import SQLAlchemy
from werkzeug.utils import secure_filename

# -----
# App configuration
# -----
app = Flask(__name__)

# SQLite DB
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

# File upload settings
app.config["UPLOAD_FOLDER"] = os.path.join(os.getcwd(), "uploads")
app.config["MAX_CONTENT_LENGTH"] = 2 * 1024 * 1024 # 2 MB max
ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg", "gif"}

# Ensure upload folder exists
os.makedirs(app.config["UPLOAD_FOLDER"], exist_ok=True)
```

```

db = SQLAlchemy(app)

# -----
# Model
# -----
class Customer(db.Model):
    __tablename__ = "customers"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120), nullable=False)
    city = db.Column(db.String(120))
    email = db.Column(db.String(120), unique=True, nullable=False)
    phone = db.Column(db.String(20), unique=True, nullable=False)
    photo_filename = db.Column(db.String(255)) # stored
    filename on disk
    photo_original_name = db.Column(db.String(255)) # original
    upload name (for reference)

    def to_dict(self):
        return {
            "id": self.id,
            "name": self.name,
            "city": self.city,
            "email": self.email,
            "phone": self.phone,
            "photo_url": url_for("get_customer_photo", id=self.id,
            _external=True) if self.photo_filename else None,
        }

# -----
# Helpers
# -----
def allowed_file(filename: str) -> bool:
    return "." in filename and filename.rsplit(".", 1)[1].lower() in
ALLOWED_EXTENSIONS

def save_uploaded_photo(file_storage):
    """
    Validates & saves an uploaded photo.
    Returns (stored_filename, original_filename).
    """
    if file_storage and file_storage.filename:
        original_name = secure_filename(file_storage.filename)
        if not allowed_file(original_name):
            abort(400, description="Invalid file type. Allowed: png,
jpg, jpeg, gif")
        ext = os.path.splitext(original_name)[1].lower()
        stored_name = f"{uuid4().hex}{ext}" # avoid collisions
        save_path = os.path.join(app.config["UPLOAD_FOLDER"],
stored_name)
        file_storage.save(save_path)

```

```

        return stored_name, original_name
    return None, None

def delete_photo_if_exists(stored_filename):
    if not stored_filename:
        return
    path = os.path.join(app.config["UPLOAD_FOLDER"], stored_filename)
    try:
        if os.path.exists(path):
            os.remove(path)
    except Exception:
        # In real apps, log this
        pass

# -----
# Routes
# -----

@app.route("/api/customers", methods=["POST"])
def create_customer():
    """
    Create a customer (multipart/form-data).
    Fields: name, city, email, phone, photo (file)
    """
    # Validate form fields
    name = request.form.get("name")
    city = request.form.get("city")
    email = request.form.get("email")
    phone = request.form.get("phone")

    if not all([name, email, phone]):
        return jsonify({"error": "name, email and phone are required"}), 400

    # Check uniqueness for email/phone (simple check; database
    uniqueness enforces too)
    if Customer.query.filter_by(email=email).first():
        return jsonify({"error": "email already exists"}), 409
    if Customer.query.filter_by(phone=phone).first():
        return jsonify({"error": "phone already exists"}), 409

    photo_file = request.files.get("photo")
    stored_name, original_name = save_uploaded_photo(photo_file)

    customer = Customer(
        name=name.strip(),
        city=(city or "").strip(),
        email=email.strip(),
        phone=phone.strip(),
        photo_filename=stored_name,
        photo_original_name=original_name,

```

```

    )
    db.session.add(customer)
    db.session.commit()

    return jsonify({"message": "created", "data":
customer.to_dict()}), 201

@app.route("/api/customers", methods=["GET"])
def list_customers():
    customers = Customer.query.order_by(Customer.id).all()
    return jsonify({"data": [c.to_dict() for c in customers]}), 200

@app.route("/api/customers/<int:id>", methods=["GET"])
def get_customer(id):
    customer = Customer.query.get_or_404(id)
    return jsonify({"data": customer.to_dict()}), 200

@app.route("/api/customers/<int:id>", methods=["PUT", "PATCH"])
def update_customer(id):
    """
    Update basic fields and optionally replace photo.
    Accepts multipart/form-data to support new photo.
    """
    customer = Customer.query.get_or_404(id)

    # These may come from either form or JSON; we keep it consistent
    with form for file support.
    name = request.form.get("name")
    city = request.form.get("city")
    email = request.form.get("email")
    phone = request.form.get("phone")

    if email and Customer.query.filter(Customer.email == email,
Customer.id != id).first():
        return jsonify({"error": "email already exists"}), 409
    if phone and Customer.query.filter(Customer.phone == phone,
Customer.id != id).first():
        return jsonify({"error": "phone already exists"}), 409

    if name: customer.name = name.strip()
    if city is not None: customer.city = city.strip()
    if email: customer.email = email.strip()
    if phone: customer.phone = phone.strip()

    # Replace photo if provided
    photo_file = request.files.get("photo")
    if photo_file and photo_file.filename:
        stored_name, original_name = save_uploaded_photo(photo_file)
        # delete old

```

```

        # delete old
        delete_photo_if_exists(customer.photo_filename)
        customer.photo_filename = stored_name
        customer.photo_original_name = original_name

    db.session.commit()
    return jsonify({"message": "updated", "data":
customer.to_dict()}), 200

@app.route("/api/customers/<int:id>", methods=["DELETE"])
def delete_customer(id):
    customer = Customer.query.get_or_404(id)
    delete_photo_if_exists(customer.photo_filename)
    db.session.delete(customer)
    db.session.commit()
    return jsonify({"message": "deleted"}), 200

@app.route("/api/customers/<int:id>/photo", methods=["GET"])
def get_customer_photo(id):
    """
    Serve the stored image file for a customer.
    """
    customer = Customer.query.get_or_404(id)
    if not customer.photo_filename:
        abort(404, description="No photo for this customer")
    return send_from_directory(app.config["UPLOAD_FOLDER"],
customer.photo_filename)

# Initialize DB
with app.app_context():
    db.create_all()

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

```

and the requirements.txt here:

```

Flask==3.0.3
Flask-SQLAlchemy==3.1.1

```

8. Key Takeaways

- Use `multipart/form-data` for file uploads.
- Validate file type and size.

- Store files safely in a dedicated folder (or cloud storage).
- Save only the **filename** in the database.
- Provide a REST endpoint to fetch the uploaded file.

CORS Handling in Flask with Flask-CORS

When you build **REST APIs with Flask**, chances are you'll need to serve requests from **different domains** — like a **React frontend (localhost:3000)** calling a **Flask backend (localhost:5000)**.

By default, browsers **block cross-origin requests** for security reasons. This is known as the **Same-Origin Policy (SOP)**.

To solve this, we use **CORS (Cross-Origin Resource Sharing)**.

In this tutorial, we'll cover:

- What CORS is and why it matters
- How Flask-CORS works
- Applying CORS globally vs per-route
- Configuring allowed origins, headers, and methods
- Handling preflight (**OPTIONS**) requests
- Example: **E-commerce product API (Indian products)**

Part 1: What is CORS?

- **Same-Origin Policy** → A web app can only call APIs from the *same* domain/port.
- **CORS** → A mechanism that tells browsers *which cross-origin requests are allowed*.

Example:

- React frontend at `http://localhost:3000`
- Flask API at `http://localhost:5000`

Without CORS → Request is blocked. With CORS → Flask responds with headers like:

```
Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Content-Type, Authorization
```

Part 2: Install Flask-CORS

```
pip install flask flask-restful flask-cors
```

Part 3: Basic Flask App

app.py

```
from flask import Flask, jsonify
from flask_restful import Api, Resource
from flask_cors import CORS

app = Flask(__name__)
api = Api(app)

# Enable CORS globally
CORS(app)

products = [
    {"id": 1, "name": "Amul Butter", "price": 50},
    {"id": 2, "name": "Tata Tea", "price": 200},
    {"id": 3, "name": "Parle-G Biscuits", "price": 10},
]

class ProductList(Resource):
    def get(self):
        return jsonify(products)

api.add_resource(ProductList, "/products")

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

Now, **any frontend** (React, Angular, Vue) can call `/products`.

Part 4: Restricting CORS to Specific Origins

Allow **only React frontend** (localhost:3000):

```
CORS(app, resources={r"/products/*": {"origins":
"http://localhost:3000"}})
```

Part 5: Applying CORS Per-Route

Instead of applying globally:

```
from flask_cors import cross_origin

class ProductDetail(Resource):
    @cross_origin(origins="http://localhost:3000")
    def get(self, product_id):
        product = next((p for p in products if p["id"] ==
product_id), None)
        if product:
            return jsonify(product)
        return {"error": "Product not found"}, 404

api.add_resource(ProductDetail, "/products/<int:product_id>")
```

Part 6: Handling Custom Headers & Methods

Allow **Authorization** headers & **PUT** method:

```
CORS(app, resources={r"/products/*": {
    "origins": "http://localhost:3000",
    "methods": ["GET", "POST", "PUT"],
    "allow_headers": ["Content-Type", "Authorization"]
}})
```

This lets the frontend send `Authorization: Bearer <token>` headers.

Part 7: Preflight Requests (OPTIONS)

Browsers send an **OPTIONS** request before `POST/PUT/DELETE`. Flask-CORS handles this automatically, returning:

```
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
```

Without Flask-CORS, you'd have to manually handle **OPTIONS**.

Part 8: Example: React Calling Flask API

React fetch call:


```
fetch('http://localhost:5000/products', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  },
})
.then((res) => res.json())
.then((data) => console.log(data));
```

With Flask-CORS → Works 🚀 Without Flask-CORS → Browser blocks request ❌

Part 9: Security Best Practices

- **Don't use CORS (app) blindly** in production → restrict origins.
- Allow only **specific domains** (your frontend, mobile apps).
- Limit **methods** (e.g., allow `GET`, `POST` but not `DELETE`).
- Limit **headers** (e.g., allow `Content-Type`, `Authorization`).

Conclusion

We've learned how to: ✅ Understand CORS & Same-Origin Policy ✅ Enable CORS globally with Flask-CORS ✅ Restrict origins, methods, and headers ✅ Handle preflight requests automatically ✅ Secure CORS in production

With these techniques, your Flask APIs can safely serve requests from **different domains and frontends** 🚀

Security Best Practices for Flask REST APIs (OWASP API Top 10)

APIs power everything — mobile apps, web apps, IoT devices, and microservices. But with great power comes **great responsibility**. A poorly secured API can expose sensitive data, open backdoors for attackers, and even take down your entire system.

To guide developers, the **OWASP Foundation** publishes the **OWASP API Security Top 10** list, which highlights the most critical risks for APIs.

In this tutorial, we'll:

- Understand the **OWASP API Top 10**
- See how vulnerabilities apply to **Flask REST APIs**
- Learn **mitigation strategies** with **code examples**
- Build a mindset for **secure API development**

API1:2019 – Broken Object Level Authorization (BOLA)

Problem: Users can access data they shouldn't (e.g., `/users/123/orders` shows another user's orders).

Fix: Always check if the **authenticated user owns the resource**.

```
@app.route("/orders/<int:order_id>")
@jwt_required()
def get_order(order_id):
    current_user = get_jwt_identity()
    order = Order.query.get(order_id)

    if not order or order.user_id != current_user["id"]:
        return {"error": "Unauthorized"}, 403

    return jsonify(order.to_dict())
```

API2:2019 – Broken User Authentication

Problem: Weak or missing authentication lets attackers impersonate users.

Fix: Use strong **JWT authentication** with refresh tokens.

```
from flask_jwt_extended import create_access_token,
create_refresh_token

@app.route("/login", methods=["POST"])
def login():
    data = request.json
    user = User.query.filter_by(username=data["username"]).first()

    if user and user.check_password(data["password"]):
        return {
            "access_token": create_access_token(identity={"id":
user.id}),
            "refresh_token": create_refresh_token(identity={"id":
user.id})
        }
    return {"error": "Invalid credentials"}, 401
```

API3:2019 – Excessive Data Exposure

Problem: Returning sensitive fields (e.g., passwords, tokens, internal IDs).

Fix: Only return necessary fields.

```
def safe_user(user):  
    return {"id": user.id, "username": user.username, "email":  
user.email}  
  
@app.route("/users/<int:user_id>")  
def get_user(user_id):  
    user = User.query.get(user_id)  
    return jsonify(safe_user(user))
```

API4:2019 – Lack of Resources & Rate Limiting

Problem: No limits → API abused with brute force or DoS.

Fix: Use **Flask-Limiter**.

```
from flask_limiter import Limiter  
from flask_limiter.util import get_remote_address  
  
limiter = Limiter(key_func=get_remote_address, app=app)  
  
@app.route("/login", methods=["POST"])  
@limiter.limit("5 per minute")  
def login():  
    ...
```

API5:2019 – Broken Function Level Authorization

Problem: Users access admin endpoints (`/admin/delete-user`) without proper role checks.

Fix: Implement **RBAC (Role-Based Access Control)**.

```
@app.route("/admin/users")  
@jwt_required()  
def list_users():  
    current_user = get_jwt_identity()  
    if current_user.get("role") != "admin":  
        return {"error": "Forbidden"}, 403  
  
    return jsonify([safe_user(u) for u in User.query.all()])
```

API6:2019 – Mass Assignment

Problem: Attackers send extra JSON fields (`role=admin`) during registration.

Fix: Explicitly whitelist allowed fields.

```
@app.route("/register", methods=["POST"])
def register():
    data = request.json
    user = User(
        username=data.get("username"),
        email=data.get("email"),
    )
    user.set_password(data.get("password"))
    db.session.add(user)
    db.session.commit()
    return {"message": "User registered successfully"}
```

API7:2019 – Security Misconfiguration

Problem: Debug mode enabled in production, exposing secrets.

Fix:

- Disable `app.debug = True` in production.
- Store secrets in environment variables.

```
import os

app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY", "fallback-secret")
```

API8:2019 – Injection (SQL/NoSQL/Command Injection)

Problem: Unsafe query building allows SQL injection.

✖ Bad:

```
user = db.session.execute(f"SELECT * FROM users WHERE id={user_id}")
```

✔ Good (SQLAlchemy ORM):

```
user = User.query.filter_by(id=user_id).first()
```

API9:2019 – Improper Assets Management

Problem: Old API versions remain public (`/api/v1/users` + `/api/v2/users`).

Fix:

- Document API versions clearly.
- Deprecate old versions.
- Use **API gateways** (Kong, Nginx, etc.) for routing.

API10:2019 – Insufficient Logging & Monitoring

Problem: Attacks go unnoticed due to lack of logs.

Fix: Log security events (but avoid sensitive data).

```
import logging
logging.basicConfig(filename="api.log", level=logging.INFO)

@app.after_request
def log_request(response):
    logging.info(f"{request.remote_addr} {request.method} {request.path} {response.status}")
    return response
```

Security Checklist (Flask APIs)

✅ Use HTTPS (TLS) everywhere ✅ Secure authentication (JWT/OAuth2/SSO) ✅ Validate user input (Marshmallow / Pydantic) ✅ Apply rate limiting & throttling ✅ Restrict CORS origins in production ✅ Implement RBAC/ABAC ✅ Encrypt sensitive data (passwords with bcrypt) ✅ Keep dependencies updated ✅ Monitor API activity

Conclusion

By applying the **OWASP API Top 10** practices, you:

- Protect your users' data
- Defend against common API attacks
- Build **trustworthy APIs** that scale securely

Security is not a one-time fix — it's a **continuous process**. Combine these best practices with **automated testing, monitoring, and audits** for maximum protection.

Flask Blueprints for Scalable APIs

When building small Flask apps, it's fine to keep all routes in a single `app.py`. But as your application grows (multiple features, dozens of endpoints), a **single file becomes messy and hard to maintain**.

This is where **Flask Blueprints** come in. They let us **organize routes by feature** and keep code modular, just like Django apps or Express.js routers.

In this tutorial, we'll build a small **Customer API** using Blueprints.

1. What are Blueprints?

Think of a Blueprint as a **mini-application** inside Flask.

- Each Blueprint can define its **own routes, error handlers**, and even **static files**.
- Later, we register these Blueprints on the main Flask app.
- This makes it easy to **split code by modules**: e.g., `customers`, `products`, `orders`.

2. Project Structure

Here's the structure we'll follow:

```
flask-blueprint-demo/
├── app.py                # Main entry point
├── extensions.py         # Extensions like SQLAlchemy
├── models.py            # Database models
├── customers/
│   ├── __init__.py
│   ├── routes.py        # Customer endpoints
│   └── service.py        # (Optional) Business logic
└── config.py            # Configurations
```

3. Setting up Flask with Config and Extensions

First, create `config.py`:

```
import os

class Config:
    SQLALCHEMY_DATABASE_URI = "sqlite:///app.db"
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.environ.get("SECRET_KEY", "dev")
```

Now create `extensions.py`:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

4. The Models

Create `models.py` for database models:

```
from extensions import db

class Customer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120), nullable=False)
    city = db.Column(db.String(120))
    email = db.Column(db.String(120), unique=True, nullable=False)
    phone = db.Column(db.String(20), unique=True, nullable=False)

    def to_dict(self):
        return {
            "id": self.id,
            "name": self.name,
            "city": self.city,
            "email": self.email,
            "phone": self.phone
        }
```

5. Creating the Customer Blueprint

Inside `customers/routes.py`:

```

from flask import Blueprint, request, jsonify
from extensions import db
from models import Customer

# Define blueprint
customers_bp = Blueprint("customers", __name__,
url_prefix="/api/customers")

@customers_bp.route("/", methods=["POST"])
def create_customer():
    data = request.json
    name, city, email, phone = data.get("name"), data.get("city"),
data.get("email"), data.get("phone")

    if not (name and email and phone):
        return jsonify({"error": "name, email, phone are required"}),
400

    if Customer.query.filter_by(email=email).first():
        return jsonify({"error": "email already exists"}), 409

    customer = Customer(name=name, city=city, email=email,
phone=phone)
    db.session.add(customer)
    db.session.commit()

    return jsonify(customer.to_dict()), 201

@customers_bp.route("/", methods=["GET"])
def list_customers():
    customers = Customer.query.all()
    return jsonify([c.to_dict() for c in customers])

@customers_bp.route("/<int:id>", methods=["GET"])
def get_customer(id):
    customer = Customer.query.get_or_404(id)
    return jsonify(customer.to_dict())

```

Concept:

- `Blueprint("customers", __name__, url_prefix="/api/customers")` → All routes in this blueprint are prefixed with `/api/customers`.
- Each route works just like a normal Flask route.

6. Registering the Blueprint in `app.py`

Now the main application just imports and registers blueprints.

```
from flask import Flask
from config import Config
from extensions import db
from customers.routes import customers_bp

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    db.init_app(app)

    # Register blueprints
    app.register_blueprint(customers_bp)

    with app.app_context():
        db.create_all()

    return app

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

Concept:

- `create_app` is an **application factory pattern** (good for scaling).
- All blueprints are registered in one place.
- The main app doesn't need to know all routes — they live in their own modules.

7. Testing the API

Let's test with **sample data (Vinod)**.

Create Customer

```
curl -X POST http://127.0.0.1:5000/api/customers/ \
  -H "Content-Type: application/json" \
  -d
  '{"name": "Vinod", "city": "Bangalore", "email": "vinod@vinod.co", "phone": "9"
```

Get All Customers

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

Response:

```
[
  {
    "id": 1,
    "name": "Vinod",
    "city": "Bangalore",
    "email": "vinod@vinod.co",
    "phone": "9731424784"
  }
]
```

8. Why Blueprints Matter

Without Blueprints, all routes would live in `app.py`. As you add features like `products`, `orders`, or `auth`, the file grows to thousands of lines.

With Blueprints:

- `customers/routes.py` → only customer APIs
- `products/routes.py` → only product APIs
- `auth/routes.py` → only auth APIs

This separation makes your API **modular, maintainable, and scalable**.

Summary

- Blueprints help split APIs into modules, making them **scalable and maintainable**.
- Each Blueprint defines its own routes and is registered in the main app.
- This pattern is essential for **large Flask applications**.

API Documentation with Swagger/OpenAPI in Flask

When building REST APIs, it's not enough to just create endpoints — you also need to **document them clearly** so that developers (and even future you) know how to use them.

That's where **Swagger / OpenAPI** comes in. It allows you to:

- Describe your API endpoints
- Specify request/response formats

- Provide a UI to test APIs interactively

In Flask, one of the easiest ways to integrate Swagger is with **Flasgger**.

1. What is Flasgger?

Flasgger is a Flask extension that:

- Reads OpenAPI/Swagger specs (YAML or JSON)
- Automatically generates a **Swagger UI** (interactive documentation)
- Lets you add documentation inline in your Flask routes

So, you write your APIs as usual, then just **annotate them with docstrings**, and Flasgger builds the documentation page.

2. Installing Dependencies

Install the required packages:

```
pip install flask flask-sqlalchemy flasgger
```

3. Project Structure

We'll keep it simple:

```
flask-swagger-demo/  
|  
├─ app.py  
├─ extensions.py  
├─ models.py  
└─ customers/  
    ├─ __init__.py  
    └─ routes.py
```

4. Setting Up Flasgger

In `app.py`:

```
from flask import Flask
from config import Config
from extensions import db
from customers.routes import customers_bp
from flasgger import Swagger

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    db.init_app(app)

    # Register blueprints
    app.register_blueprint(customers_bp)

    # Initialize Swagger
    swagger = Swagger(app)

    with app.app_context():
        db.create_all()

    return app

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

Now, when you run the app and visit <http://127.0.0.1:5000/apidocs/>, you'll see a Swagger UI (though empty for now).

5. Adding Swagger Docs to Endpoints

Let's document the **Customer API**.

In `customers/routes.py`:

```
from flask import Blueprint, request, jsonify
from extensions import db
from models import Customer
from flasgger.utils import swag_from

customers_bp = Blueprint("customers", __name__,
url_prefix="/api/customers")

@customers_bp.route("/", methods=["POST"])
@swag_from({
    "tags": ["Customers"],
```



```

db.session.commit()

return jsonify(customer.to_dict()), 201

@customers_bp.route("/", methods=["GET"])
@swag_from({
    "tags": ["Customers"],
    "description": "List all customers",
    "responses": {
        200: {
            "description": "A list of customers",
            "examples": {
                "application/json": [
                    {
                        "id": 1,
                        "name": "Vinod",
                        "city": "Bangalore",
                        "email": "vinod@vinod.co",
                        "phone": "9731424784"
                    }
                ]
            }
        }
    }
})
def list_customers():
    """Endpoint to list customers"""
    customers = Customer.query.all()
    return jsonify([c.to_dict() for c in customers])

```

Concepts here:

- `@swag_from({...})` → Inline YAML/JSON definition for the route.
- `"tags"` → Group APIs by category (e.g., "Customers").
- `"parameters"` → Define inputs (query params, body, headers, etc.).
- `"responses"` → Define possible status codes and example responses.

6. Trying It Out

Run the app:

```
python app.py
```

Open <http://127.0.0.1:5000/apidocs/>. You'll now see:

- A **"Customers"** section

- `POST /api/customers/` with request body schema
- `GET /api/customers/` with example responses

You can **try out APIs directly from Swagger UI** — super useful for testing and demos!

7. Testing with Sample Data (Vinod)

Try the **POST /api/customers/** endpoint in Swagger UI with:

```
{
  "name": "Vinod",
  "city": "Bangalore",
  "email": "vinod@vinod.co",
  "phone": "9731424784"
}
```

Then use **GET /api/customers/** → You'll see the saved customer listed.

8. Benefits of Swagger/OpenAPI

- **Interactive docs:** Consumers can test APIs directly.
- **Clear contract:** Defines inputs, outputs, and errors.
- **Client code generation:** Tools can auto-generate API clients in Python, Java, JS, etc.
- **Team collaboration:** Frontend/backend teams can work in parallel with agreed contracts.

9. Full Example Code (for reference)

app.py

```
from flask import Flask
from config import Config
from extensions import db
from customers.routes import customers_bp
from flasgger import Swagger

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    db.init_app(app)
    app.register_blueprint(customers_bp)

    swagger = Swagger(app)

    with app.app_context():
        db.create_all()

    return app

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

customers/routes.py → (as shown above with @swag_from)

models.py

```
from extensions import db

class Customer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120), nullable=False)
    city = db.Column(db.String(120))
    email = db.Column(db.String(120), unique=True, nullable=False)
    phone = db.Column(db.String(20), unique=True, nullable=False)

    def to_dict(self):
        return {
            "id": self.id,
            "name": self.name,
            "city": self.city,
            "email": self.email,
            "phone": self.phone
        }
```

extensions.py


```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

config.py

```
class Config:
    SQLALCHEMY_DATABASE_URI = "sqlite:///app.db"
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = "dev"
```

10. Summary

- Swagger/OpenAPI is essential for documenting REST APIs.
- With **Flasgger**, you can add docs using decorators and YAML/JSON definitions.
- The result is a **beautiful interactive API explorer** at </apidocs/>.
- This improves developer experience and accelerates adoption of your API.

Testing Flask REST APIs with Pytest (Unit & Integration)

When building REST APIs, it's not enough to just write endpoints — you need to **test them thoroughly**. Testing ensures:

- Your APIs return the expected responses.
- New changes don't break existing functionality.
- Edge cases and errors are handled gracefully.

In Python, one of the most popular testing frameworks is **Pytest**. In this tutorial, we'll use Pytest to write **unit tests** (testing functions in isolation) and **integration tests** (testing API endpoints via HTTP).

1. Setting Up the Project

We'll build on our **Customer API** example. Make sure you have these installed:

```
pip install flask flask-sqlalchemy pytest pytest-flask
```

2. What are Unit Tests vs Integration Tests?

- **Unit Tests** → Test individual components (e.g., database model methods).

- **Integration Tests** → Test the full flow by calling the API endpoints and checking responses.

In Flask:

- Unit test = call Python functions directly.
- Integration test = use Flask's test client to simulate API requests (`client.get()`, `client.post()`, etc.).

3. Project Structure for Tests

```
flask-testing-demo/  
├── api/  
│   ├── app.py  
│   ├── models.py  
│   ├── customers/  
│   │   ├── __init__.py  
│   │   └── routes.py  
│   ├── extensions.py  
│   └── config.py  
└── tests/  
    ├── conftest.py  
    ├── test_models.py  
    └── test_customers_api.py
```

4. Pytest Fixtures for Test Setup

In `tests/conftest.py` we define **fixtures** (reusable setup code):

```

import pytest
from api.app import create_app
from api.extensions import db

@pytest.fixture
def app():
    """Create and configure a new app instance for each test"""
    app = create_app()
    app.config.update({
        "TESTING": True,
        "SQLALCHEMY_DATABASE_URI": "sqlite:///memory:", # in-memory
        "SQLALCHEMY_TRACK_MODIFICATIONS": False
    })

    with app.app_context():
        db.create_all()
        yield app
        db.drop_all()

@pytest.fixture
def client(app):
    """A test client for the app"""
    return app.test_client()

@pytest.fixture
def runner(app):
    """A CLI runner for the app"""
    return app.test_cli_runner()

```

Concept:

- `pytest.fixture` → Creates reusable components for tests.
- Here, `app` creates a **new Flask app** with an **in-memory SQLite DB** for each test (so tests don't affect your real DB).
- `client` provides a fake HTTP client to call your APIs.

5. Writing Unit Tests for the Model

In `tests/test_models.py`:

```
from api.models import Customer
from api.extensions import db

def test_create_customer(app):
    """Test creating and saving a Customer model"""
    with app.app_context():
        customer = Customer(
            name="Vinod",
            city="Bangalore",
            email="vinod@vinod.co",
            phone="9731424784"
        )
        db.session.add(customer)
        db.session.commit()

        saved =
Customer.query.filter_by(email="vinod@vinod.co").first()
        assert saved is not None
        assert saved.name == "Vinod"
```

Concept: This test **bypasses the API** and directly checks if the `Customer` model works correctly with the DB.

6. Writing Integration Tests for the API

In `tests/test_customers_api.py`:

```
def test_create_customer(client):
    """Test POST /api/customers/"""
    response = client.post("/api/customers/", json={
        "name": "Vinod",
        "city": "Bangalore",
        "email": "vinod@vinod.co",
        "phone": "9731424784"
    })

    assert response.status_code == 201
    data = response.get_json()
    assert data["name"] == "Vinod"
    assert data["email"] == "vinod@vinod.co"

def test_list_customers(client):
    """Test GET /api/customers/"""
    # First, create a customer
    client.post("/api/customers/", json={
        "name": "Vinod",
        "city": "Bangalore",
        "email": "vinod@vinod.co",
        "phone": "9731424784"
    })

    response = client.get("/api/customers/")
    assert response.status_code == 200
    data = response.get_json()
    assert len(data) == 1
    assert data[0]["city"] == "Bangalore"
```

Concept:

- `client.post()` simulates an API call.
- We check both **status codes** and **response data**.
- This ensures the API works end-to-end.

7. Running the Tests

Run tests with:

```
PYTHONPATH=. pytest -v
```

Sample output:

```
tests/test_models.py::test_create_customer PASSED
tests/test_customers_api.py::test_create_customer PASSED
tests/test_customers_api.py::test_list_customers PASSED
```

8. Key Benefits of Testing with Pytest

✅ Catches bugs early ✅ Provides confidence when refactoring ✅ Makes your API predictable and stable ✅ Helps onboard new developers faster

You now have:

- `tests/conftest.py` → fixtures
- `tests/test_models.py` → unit test
- `tests/test_customers_api.py` → integration tests

Together, they cover **models + API routes**.

10. Summary

- Use **Pytest fixtures** to spin up a clean test app & DB.
- Write **unit tests** to test models in isolation.
- Write **integration tests** to test full API flows.
- Run tests with `pytest` and watch the green ✅ appear!

API Gateway Integration with Kong

When you deploy multiple microservices or APIs, managing them directly can get messy:

- How do you authenticate requests?
- How do you rate limit?
- How do you centralize logging/monitoring?

This is where an **API Gateway** comes in.

An API Gateway sits between your clients (frontend/mobile) and backend services. It provides:

✅ Centralized access to APIs ✅ Authentication & authorization ✅ Rate limiting, logging, caching ✅ Request/response transformation

One popular open-source gateway is **Kong**.

What is Kong Gateway?

Kong is a **cloud-native, open-source API gateway** built on Nginx. It helps manage, monitor, and secure APIs.

With Kong you can:

- Register APIs (called **Services** in Kong).
- Expose them via **Routes**.
- Enhance them with **Plugins** (auth, rate limiting, CORS, logging, etc.).

Managing multiple APIs in a microservice architecture can get messy. Instead of exposing each service individually, we use an **API Gateway**.

In this tutorial we'll set up:

- Two Flask APIs (Products & Customers).
- Kong Gateway in front of them.
- Declarative routing (`kong.yml`) so we don't need to configure routes manually.

The final setup:

- `http://localhost:8000/api/products` → Products API
- `http://localhost:8000/api/customers` → Customers API

1. Project Structure

```
kong-flask-gateway/  
├──  
├── docker-compose.yml  
├── kong.yml  
├── products/  
│   ├── app.py  
│   └── Dockerfile  
└── customers/  
    ├── app.py  
    └── Dockerfile
```

2. Flask Services

products/app.py

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/api/products")
def get_products():
    return jsonify([
        {"id": 1, "name": "Laptop", "price": 50000},
        {"id": 2, "name": "Mobile", "price": 20000}
    ])

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=6010)
```

customers/app.py

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/api/customers")
def get_customers():
    return jsonify([
        {"id": 1, "name": "Vinod", "city": "Bangalore", "email":
"vinod@vinod.co", "phone": "9731424784"},
        {"id": 2, "name": "John", "city": "Delhi", "email":
"john@example.com", "phone": "9812345678"}
    ])

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=6020)
```

3. Dockerfile for Flask APIs

Both services use the same simple Dockerfile.

Example: **products/Dockerfile**


```
FROM python:3.10-slim

WORKDIR /app
COPY app.py /app
RUN pip install flask

CMD ["python", "app.py"]
```

(Same Dockerfile works for `customers/`, just copy it into that folder.)

4. Kong Declarative Config (`kong.yml`)

Instead of running `curl` commands against the Admin API, we use a **DB-less declarative config**:

```
_format_version: '3.0'

services:
  - name: products-service
    url: http://products:6010
    routes:
      - name: products-route
        paths:
          - /api/products
        strip_path: false # 🐡 preserve path so Flask sees /api/products

  - name: customers-service
    url: http://customers:6020
    routes:
      - name: customers-route
        paths:
          - /api/customers
        strip_path: false # 🐡 preserve path so Flask sees /api/customers
```

Why `strip_path: false`? By default, Kong strips the route prefix before forwarding. So `/api/customers` would be forwarded as `/` to Flask — leading to a **404**. Setting `strip_path: false` ensures Kong forwards the **full path**.

5. Docker Compose (`docker-compose.yml`)

Here's the full setup:

```
version: '3.8'

networks:
  kong-net:

services:
  products:
    build: ./products
    container_name: products-service
    networks:
      - kong-net
    ports:
      - '6010:6010'

  customers:
    build: ./customers
    container_name: customers-service
    networks:
      - kong-net
    ports:
      - '6020:6020'

  kong:
    image: kong:3
    container_name: kong
    environment:
      KONG_DATABASE: 'off' # DB-less mode
      KONG_DECLARATIVE_CONFIG: /usr/local/kong/declarative/kong.yml
      KONG_PROXY_ACCESS_LOG: /dev/stdout
      KONG_ADMIN_ACCESS_LOG: /dev/stdout
      KONG_PROXY_ERROR_LOG: /dev/stderr
      KONG_ADMIN_ERROR_LOG: /dev/stderr
      KONG_ADMIN_LISTEN: 0.0.0.0:8001
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    ports:
      - '8000:8000' # Public Gateway
      - '8001:8001' # Admin API
    networks:
      - kong-net
    depends_on:
      - products
      - customers
```

6. Run the System

Build and start everything:

```
docker compose up --build -d
```

Services:

- Products API → <http://products:6010/api/products> (internal)
- Customers API → <http://customers:6020/api/customers> (internal)
- Kong Admin API → <http://localhost:8001>
- Kong Public Proxy → <http://localhost:8000>

7. Test the Gateway

Instead of calling services directly, test via Kong:

```
curl http://localhost:8000/api/products
```

```
[
  { "id": 1, "name": "Laptop", "price": 50000 },
  { "id": 2, "name": "Mobile", "price": 20000 }
]
```

```
curl http://localhost:8000/api/customers
```

```
[
  {
    "id": 1,
    "name": "Vinod",
    "city": "Bangalore",
    "email": "vinod@vinod.co",
    "phone": "9731424784"
  },
  {
    "id": 2,
    "name": "John",
    "city": "Delhi",
    "email": "john@example.com",
    "phone": "9812345678"
  }
]
```

✅ Both APIs now go **through Kong Gateway**.

8. Inspect Kong (Optional)

You can check what Kong loaded:

```
curl http://localhost:8001/services
curl http://localhost:8001/routes
```

✅ Summary

- We created **two Flask microservices** (Products & Customers).
- Ran them via **Docker Compose**.
- Put **Kong Gateway** in front of them.
- Used **declarative DB-less config (kong.yml)** to define routes.
- Fixed the **strip_path issue** so Flask services work correctly.
- Accessed all APIs via **one single gateway**: `http://localhost:8000/api/....`

This setup is production-ready for local development. To scale, simply add more services in `kong.yml` and Kong will route them automatically.

Strategies for Upgrading Deployed APIs (Without Breaking Clients)

When APIs are deployed and in use by clients (mobile apps, partner integrations, frontends), **changes can be dangerous**. If you remove or modify endpoints abruptly, clients may break.

That's why upgrading APIs requires **careful planning and strategies**.

In this tutorial, we'll explore:

1. Why upgrading APIs is tricky
2. Strategies for safe upgrades
3. Demo with a Flask REST API

1. Why Upgrading APIs is Tricky?

Imagine we have a `Customer API`:

```
{
  "id": 1,
  "name": "Vinod",
  "city": "Bangalore",
  "email": "vinod@vinod.co",
  "phone": "9731424784"
}
```

Now let's say:

- You want to **rename** `city` → `location`
- You want to **add** `photo` URL
- You want to **remove** `phone` number

If you simply change the API, existing apps that expect `city` or `phone` will break. 🙅 That's why we need **upgrade strategies**.

2. API Upgrade Strategies

Strategy 1: Versioning (most common)

- Add a version prefix: `/api/v1/customers`, `/api/v2/customers`.
- Old clients keep using **v1**, new clients switch to **v2**.
- Both can run in parallel until v1 is deprecated.

👉 Example in Flask:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/api/v1/customers")
def customers_v1():
    return jsonify([{"id": 1, "name": "Vinod", "city": "Bangalore",
"email": "vinod@vinod.co", "phone": "9731424784"}])

@app.route("/api/v2/customers")
def customers_v2():
    return jsonify([{"id": 1, "name": "Vinod", "location":
"Bangalore", "email": "vinod@vinod.co", "photo":
"http://example.com/vinod.jpg"}])
```

✅ Clients can choose when to migrate.

Strategy 2: Backward Compatibility

- Keep supporting old fields until clients migrate.
- Add new fields alongside old ones.
- Deprecate gradually.

👉 Example:

```
@app.route("/api/customers")
def customers():
    return jsonify([
        {
            "id": 1,
            "name": "Vinod",
            "city": "Bangalore",          # old field (deprecated soon)
            "location": "Bangalore",     # new field
            "email": "vinod@vinod.co",
            "phone": "9731424784",       # will be removed later
            "photo": "http://example.com/vinod.jpg"
        }
    ])
```

✅ Clients won't break; they get both fields until you announce deprecation.

Strategy 3: Feature Flags / Query Parameters

- Allow clients to "opt-in" to new fields.
- Example: `GET /api/customers?version=2`.

👉 Example:

```
from flask import request

@app.route("/api/customers")
def customers_with_param():
    version = request.args.get("version", "1")
    if version == "2":
        return jsonify([{"id": 1, "name": "Vinod", "location":
            "Bangalore", "email": "vinod@vinod.co", "photo":
            "http://example.com/vinod.jpg"}])
    else:
        return jsonify([{"id": 1, "name": "Vinod", "city":
            "Bangalore", "email": "vinod@vinod.co", "phone": "9731424784"}])
```

✅ Useful for gradual rollouts.

Strategy 4: Deprecation Notices

- Announce breaking changes clearly (docs, headers).

- Send warnings in API responses.

👉 Example (add a custom header):

```
from flask import make_response

@app.route("/api/customers/deprecated")
def deprecated_customers():
    response = make_response(jsonify([{"id": 1, "name": "Vinod"}]))
    response.headers["Deprecation"] = "true"
    response.headers["Sunset"] = "2025-12-31" # when it will stop
    working
    return response
```

Clients see headers and know when to upgrade.

Strategy 5: API Gateway Routing

If you use an API Gateway (like Kong), you can:

- Run multiple API versions behind it.
- Route requests based on version path/header.
- Apply policies to gradually phase out old APIs.

3. Demo: Putting it Together

Let's say we're upgrading the **Customer API**:

- **v1** → **city** and **phone**
- **v2** → **location** and **photo**

Both run in parallel:

```
@app.route("/api/v1/customers")
def customers_v1():
    return jsonify([{"id": 1, "name": "Vinod", "city": "Bangalore",
"email": "vinod@vinod.co", "phone": "9731424784"}])

@app.route("/api/v2/customers")
def customers_v2():
    return jsonify([{"id": 1, "name": "Vinod", "location":
"Bangalore", "email": "vinod@vinod.co", "photo":
"http://example.com/vinod.jpg"}])
```

Clients using **v1** will not break. New clients can migrate to **v2**.

Later, you announce:

- v1 will be deprecated on **2025-12-31**.
- v2 will be the only supported version from 2026.

4. Best Practices

- **Always communicate changes early** (docs, emails, headers).
- **Keep old versions alive** for a reasonable time.
- **Document differences clearly** between versions.
- **Automate testing** for both old & new versions.
- Use an **API Gateway** for easier version routing.

5. Summary

Upgrading deployed APIs without breaking clients requires strategy:

- Use **versioning** to run old & new APIs in parallel.
- Keep **backward compatibility** where possible.
- Provide **feature flags** for gradual rollout.
- Send **deprecation notices** in responses.
- Use an **API Gateway** to manage multiple versions cleanly.

By following these strategies, you ensure smooth API upgrades — and happy clients 🎯