Web Technology / Full Stack Development

API Integration • Microservices Overview • API Documentation using Swagger / OpenAPI

1. API Integration

1.1 Introduction

API integration means connecting an application to another application or service through its API so that the two can exchange data and use each other's functions automatically. Instead of building every feature from scratch, a developer calls ready-made services – payment processing, maps, SMS/OTP, e-mail, weather, social login, cloud storage – over the Internet, usually using HTTP and JSON.

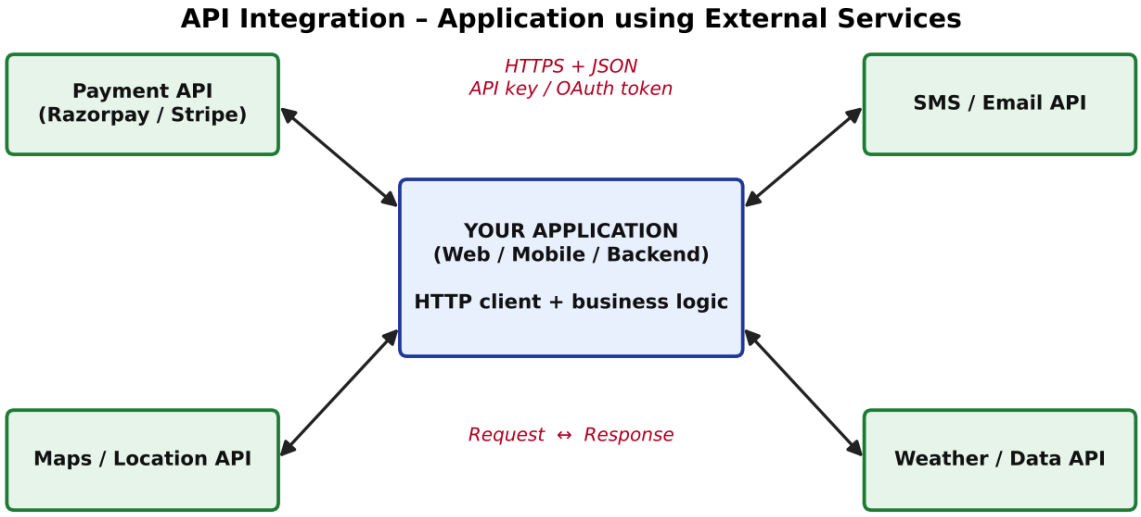


Fig. 1 API integration – an application using external services

Why integrate APIs?

- Saves development time and cost by reusing existing, tested services.
- Allows different systems (ERP, CRM, web, mobile) to share data in real time.
- Adds advanced features such as payments, maps and AI without owning the infrastructure.
- Supports automation of business workflows.

1.2 Types of APIs

By audience

Type	Description	Example
Public (Open) API	Available to any developer, often with a free tier and key	Weather, Google Maps, GitHub API

Type	Description	Example
Partner API	Shared only with selected business partners under agreement	Airline booking API for travel agents
Private (Internal) API	Used only inside an organisation	Microservices of one company
Composite API	Combines several API calls into one request	API gateway aggregating user + order data

By style / protocol

Style	Features	Typical use
REST	HTTP + JSON, resource URIs, stateless	Most web and mobile APIs
SOAP	XML envelope, WSDL, WS-Security	Banking, legacy enterprise systems
GraphQL	Single endpoint; client chooses the fields it needs	Complex front ends, mobile apps
gRPC	Binary Protocol Buffers over HTTP/2, very fast	Service-to-service calls in microservices
WebSocket	Persistent two-way connection	Chat, live scores, notifications

1.3 Steps in API Integration

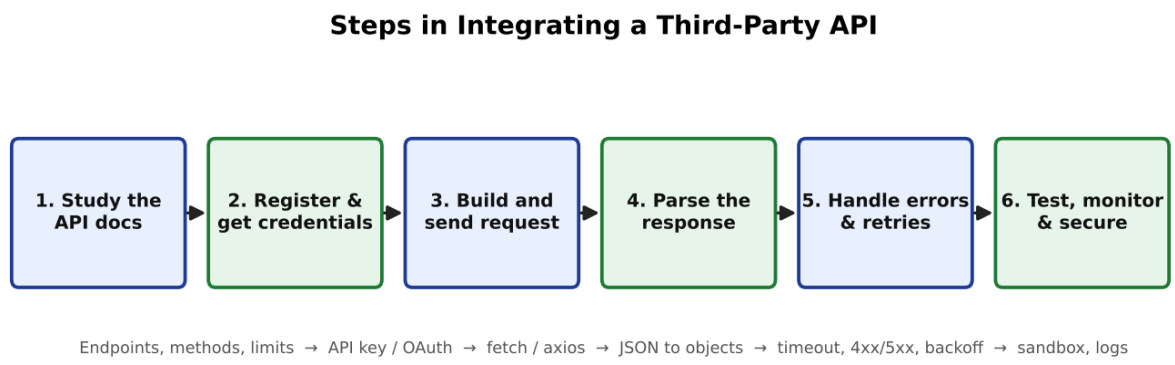


Fig. 2 Steps in integrating a third-party API

1. **Study the documentation:** find the base URL, endpoints, methods, request/response formats, rate limits and pricing.
2. **Register and obtain credentials:** create an account, get an API key or OAuth client id/secret; use the sandbox (test) environment first.
3. **Build and send the request:** set the method, URL, query parameters, headers (Authorization, Content-Type) and JSON body using fetch, axios or an SDK.
4. **Parse and map the response:** check the status code, convert JSON to objects and map the fields to your own data model.

5. **Handle errors and failures:** timeouts, 4xx/5xx codes, rate limits (429), retries with back-off, fall-back behaviour.
6. **Test, secure and monitor:** unit/integration tests with mocks, keep keys in environment variables, log calls and watch latency and error rates.

1.4 Authentication in API Integration

Method	How it works	Remarks
API key	A secret string sent in a header (x-api-key) or query parameter	Simple; identifies the application, not the user
HTTP Basic	Base64 of username:password in the Authorization header	Must be used over HTTPS only
Bearer token / JWT	Authorization: Bearer <token> obtained after login	Stateless; tokens expire
OAuth 2.0	User grants limited access to a client app without sharing the password; the server issues access tokens	Standard for 'Login with Google', GitHub, etc.
HMAC signature	Request signed with a shared secret; server recomputes and compares	Used for webhooks and financial APIs
Mutual TLS	Both client and server present certificates	High-security B2B integration

OAuth 2.0 – Authorization Code Flow

OAuth 2.0 is an authorization framework that lets a third-party application obtain limited access to a user's resources on another service. It involves four roles:

Role	Description
Resource Owner	The user who owns the data
Client	The application requesting access
Authorization Server	Authenticates the user and issues tokens (e.g. Google accounts)
Resource Server	The API that holds the protected data and accepts access tokens

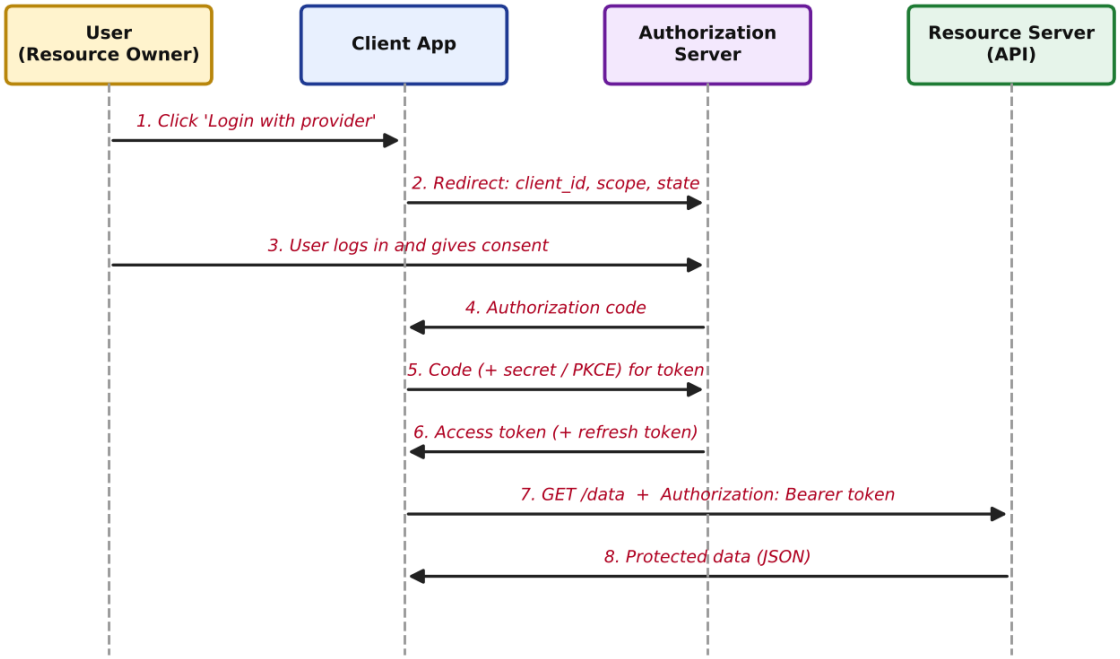


Fig. 3 OAuth 2.0 authorization code flow

The client never sees the user's password. It receives a short-lived access token (and optionally a refresh token to obtain new access tokens). Modern clients add PKCE (Proof Key for Code Exchange) to protect the code exchange, especially for single-page and mobile apps. Other grant types: client credentials (machine-to-machine), device code and refresh token.

1.5 Calling APIs from JavaScript

(a) fetch with async / await

```
async function getStudents() {
  const res = await fetch("https://api.example.com/v1/students?dept=CSE", {
    headers: { Authorization: `Bearer ${token}`, Accept: "application/json"
},
  });
  if (!res.ok) throw new Error(`HTTP ${res.status}`);   // fetch does NOT
reject on 4xx/5xx
  return res.json();
}
```

(b) POST with a JSON body

```
const res = await fetch("https://api.example.com/v1/students", {
  method: "POST",
  headers: { "Content-Type": "application/json", Authorization: `Bearer
${token}` },
  body: JSON.stringify({ name: "Anu", dept: "IT" }),
});
const created = await res.json();      // 201 Created
```

(c) axios (npm install axios)

```
const axios = require("axios");
```

```
const api = axios.create({
  baseURL: "https://api.example.com/v1",
  timeout: 5000,                                    // 5 seconds
  headers: { Authorization: `Bearer ${process.env.API_TOKEN}` },
});

try {
  const { data } = await api.get("/students", { params: { dept: "CSE", page:
1 } });
  console.log(data);
} catch (err) {
  if (err.response) console.error(err.response.status, err.response.data);
// server replied with error
  else console.error("No response / timeout", err.message);
}
```

fetch	axios
Built into browsers and Node.js 18+	Third-party library
Does not reject on HTTP error codes; manual res.ok check	Rejects on 4xx/5xx automatically
Manual JSON parsing: res.json()	Automatic JSON parsing (response.data)
Timeout via AbortSignal	Built-in timeout option, interceptors

1.6 Using an External API through Your Own Backend

API keys must never be placed in front-end (browser) code because anyone can read them. The front end calls your own Express server, which adds the secret key and forwards the request to the external API. This also lets you cache results, validate input and hide the provider's response format.

```
// .env :  WEATHER_KEY=abc123
const express = require("express");
require("dotenv").config();
const app = express();

app.get("/api/weather/:city", async (req, res) => {
  try {
    const url = `https://api.example-
weather.com/v1/current?city=${encodeURIComponent(req.params.city)}`;
    const r = await fetch(url, { headers: { "x-api-key":
process.env.WEATHER_KEY } });
    if (!r.ok) return res.status(502).json({ error: "Weather service failed"
});
    const d = await r.json();
    res.json({ city: d.city, temp: d.temperature, condition: d.summary });
// map to own format
  } catch (e) {
    res.status(504).json({ error: "Weather service unreachable" });
  }
});
app.listen(3000);
```

The URL above is a placeholder; replace it with the real provider's endpoint and header names from its documentation.

1.7 Handling Errors, Rate Limits and Retries

Situation	Recommended handling
Network error / timeout	Set a timeout; retry a limited number of times
429 Too Many Requests	Wait for the Retry-After header; slow down; queue requests
5xx server errors	Retry with exponential back-off (0.5 s, 1 s, 2 s ...) and jitter
4xx client errors	Do not retry; fix the request (401 refresh token, 400/422 correct data)
Duplicate POST on retry	Send an Idempotency-Key header so the provider processes the request once
Large result sets	Follow pagination links (page, limit, next cursor)
Provider is down	Circuit breaker, cached data or a fallback response

```
async function fetchWithRetry(url, options = {}, retries = 3) {
  for (let attempt = 0; ; attempt++) {
    try {
      const res = await fetch(url, { ...options, signal:
AbortSignal.timeout(5000) });
      if (res.ok) return await res.json();
      if (res.status !== 429 && res.status < 500) {
        const e = new Error(`HTTP ${res.status}`);
        e.fatal = true; throw e;                    // 4xx: do not retry
      }
    } catch (err) {
      if (err.fatal || attempt >= retries) throw err;
    }
    if (attempt >= retries) throw new Error("Service unavailable");
    await new Promise(r => setTimeout(r, 2 ** attempt * 500));   // 0.5 s, 1
s, 2 s
  }
}
```

1.8 Polling vs Webhooks

Sometimes the data you need is produced later – for example a payment completes after the customer authorises it. There are two ways to learn about such changes.

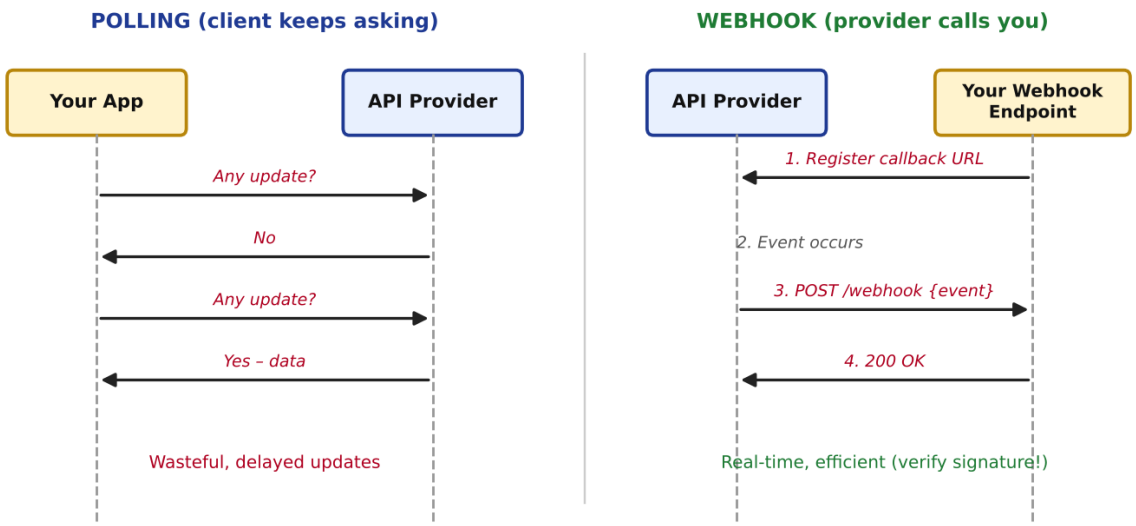


Fig. 4 Polling versus webhook

Polling	Webhook
Your app repeatedly calls the API to ask for updates	The provider sends an HTTP POST to your URL when an event occurs
Simple, but wastes requests and delays updates	Real-time and efficient
Works behind firewalls	Needs a public HTTPS endpoint
Example: check order status every 30 s	Example: payment-success event from a gateway

A webhook endpoint must verify that the request really came from the provider, normally by checking an HMAC signature, and must respond quickly with 200 OK (heavy work is queued).

```
const crypto = require("crypto");

// raw body is needed to compute the signature
app.post("/webhooks/payment", express.raw({ type: "application/json" }),
(req, res) => {
  const sig = req.get("X-Signature") || "";
  const expected = crypto.createHmac("sha256", process.env.WEBHOOK_SECRET)
                         .update(req.body).digest("hex");
  const ok = sig.length === expected.length &&
             crypto.timingSafeEqual(Buffer.from(sig), Buffer.from(expected));
  if (!ok) return res.sendStatus(401);              // not from the provider

  const event = JSON.parse(req.body);
  if (event.type === "payment.success") { /* mark order as paid */ }
  res.sendStatus(200);
});
```

1.9 CORS (Cross-Origin Resource Sharing)

Browsers apply the same-origin policy: a script loaded from one origin (scheme + host + port) cannot read responses from a different origin unless that server allows it. CORS is the mechanism by which a server tells the browser which origins may access it, using headers such

as Access-Control-Allow-Origin. For non-simple requests (e.g. JSON POST, custom headers) the browser first sends a pre-flight OPTIONS request.

```
const cors = require("cors");
app.use(cors({ origin: "https://myapp.example.com", methods: ["GET", "POST",
"PUT", "DELETE"] }));
```

CORS is enforced only by browsers, so server-to-server calls (Node.js to an external API) are not affected. Using origin: "*" with credentials is unsafe; list trusted origins instead.

1.10 Integration Patterns

Pattern	Description	Example
Point-to-point	Two systems connected directly	Website calls payment gateway
API gateway / aggregator	One entry point calls several APIs and merges results	Mobile backend combining user, order and shipping data
Batch / ETL	Large data moved on a schedule	Nightly sync of sales data to a warehouse
Event-driven	Systems communicate through events on a message broker	Order-created event consumed by inventory and e-mail services
iPaaS	Cloud integration platforms with ready connectors	Zapier, MuleSoft, Make

1.11 Challenges and Best Practices

Challenge	Best practice
API changes / versions	Pin to a version (/v1); watch deprecation notices; wrap the API in your own adapter class
Security of keys and tokens	Store in environment variables or a secrets manager; never commit to Git; rotate regularly
Different data formats	Map external fields to an internal model in one place
Latency and downtime	Timeouts, caching, retries, circuit breaker, asynchronous processing
Rate limits and cost	Cache responses, batch calls, monitor usage
Testing	Use the sandbox, mock servers (WireMock, Mockoon, Prism) and contract tests
Debugging	Log request id, status and latency (never log secrets); try endpoints in Postman, Insomnia or curl first

2. Microservices Overview

2.1 Definition

Microservices is an architectural style in which a large application is built as a collection of small, independent services. Each service implements one business capability (for example user management, orders or payments), runs in its own process, owns its own data, is deployed independently and communicates with other services through lightweight mechanisms such as HTTP/REST, gRPC or messaging.

Characteristics

- **Single responsibility:** each service does one job well (bounded context).
- **Independent deployment:** a service can be changed, tested and released without redeploying others.
- **Decentralised data:** database per service; no shared tables.
- **Technology freedom (polyglot):** each team may choose its own language and database.
- **Smart endpoints, dumb pipes:** logic is in services; communication is simple REST or messaging.
- **Resilience and scalability:** failure of one service is contained; only the busy services are scaled.
- **Automation:** CI/CD, containers and infrastructure as code are essential.

2.2 Monolithic vs Microservices Architecture

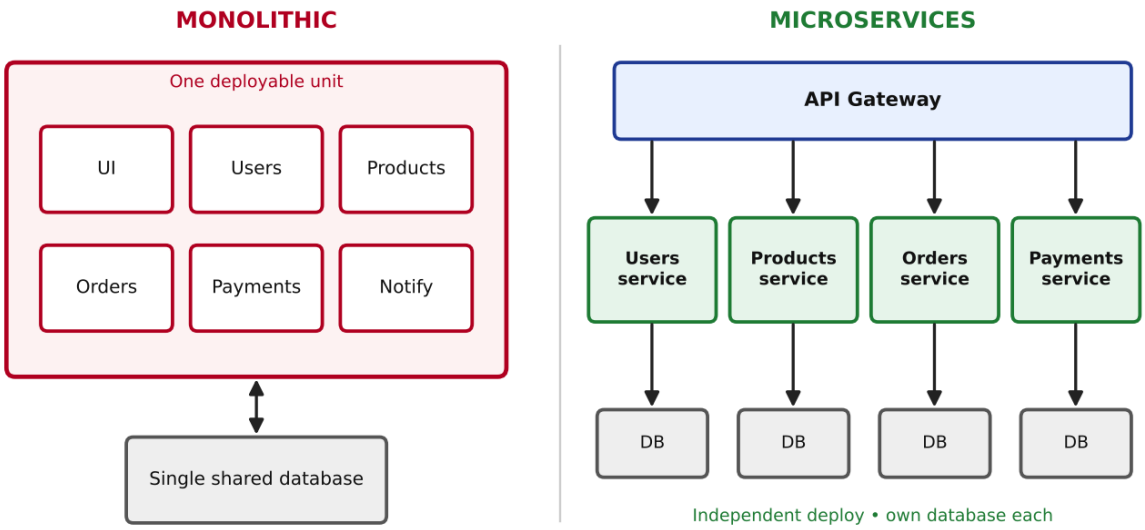


Fig. 5 Monolithic vs microservices architecture

Monolithic	Microservices
Single codebase and single deployable unit	Many small services, each deployed separately
One shared database	Database per service

Monolithic	Microservices
Whole application is scaled together	Each service scaled independently
One technology stack	Different stacks per service possible
Simple to develop, test and deploy at the beginning	Complex: distributed system, networking, monitoring
A bug or crash can bring down everything	Failures are isolated to one service
Slow releases as the code base grows	Fast, frequent releases by small teams
In-process method calls (fast)	Network calls (slower, can fail)

2.3 Microservices Architecture and Components

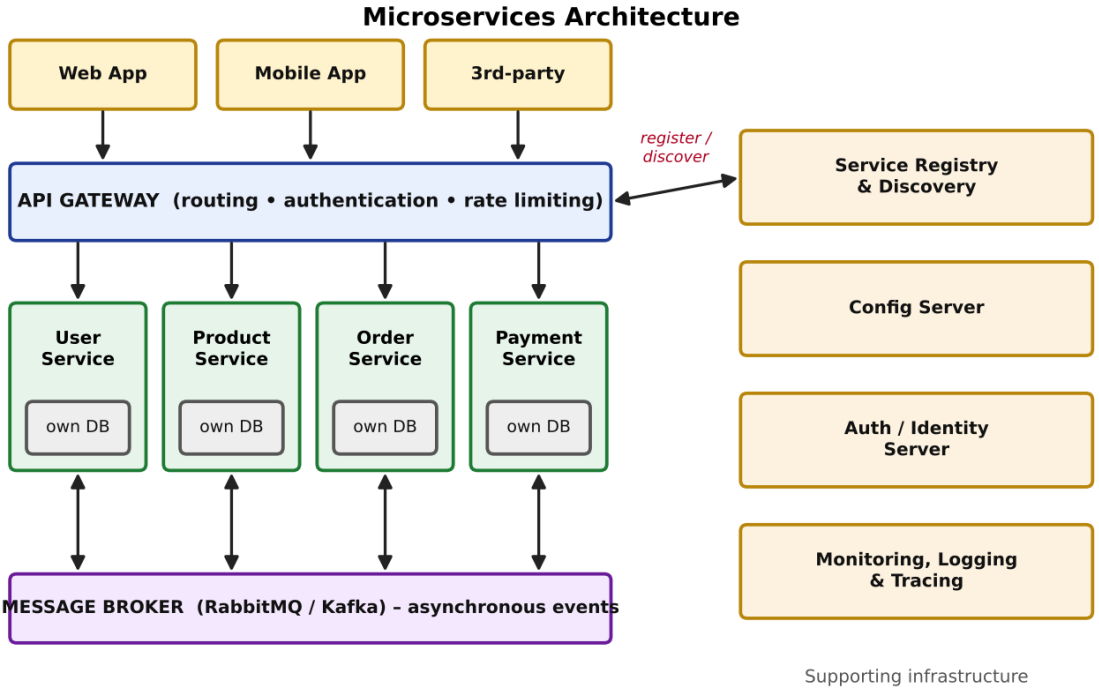


Fig. 6 Microservices architecture

Component	Role
API Gateway	Single entry point for clients; routes requests, performs authentication, rate limiting, SSL termination and response aggregation (Kong, NGINX, Spring Cloud Gateway, Express Gateway)
Microservices	Independent services, each with its own REST/gRPC API and its own database
Service Registry & Discovery	Services register their address; others look them up (Consul, Eureka, Kubernetes DNS)
Load Balancer	Distributes calls among instances of a service
Configuration Server	Central place for environment-specific settings and secrets

Component	Role
Message Broker	Asynchronous communication via queues/topics (RabbitMQ, Apache Kafka)
Auth / Identity Server	Issues and validates tokens (Keycloak, Auth0, OAuth 2.0 / JWT)
Containers & Orchestrator	Docker packages each service; Kubernetes schedules, scales and heals them
Monitoring, Logging, Tracing	Prometheus + Grafana, ELK stack, Jaeger / OpenTelemetry

2.4 Communication between Microservices

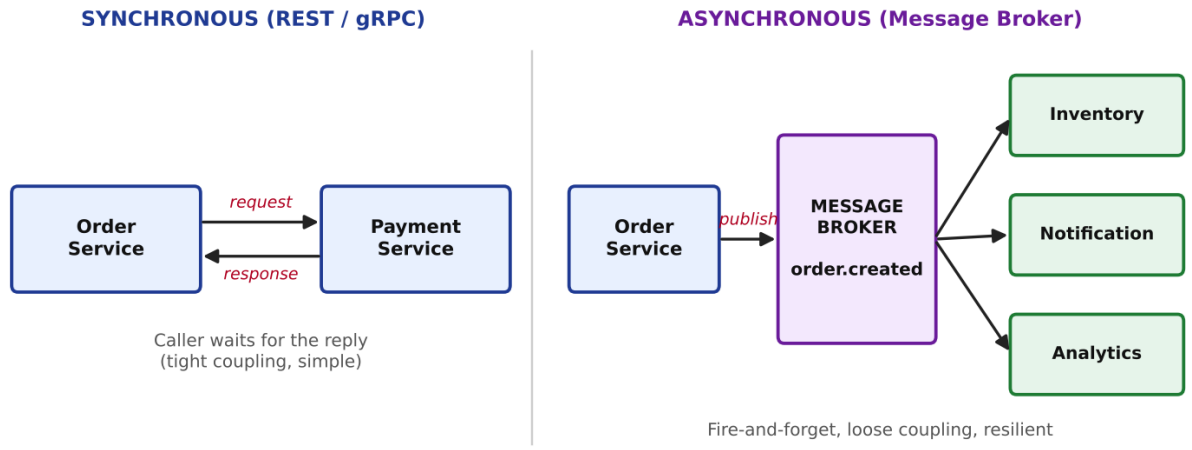


Fig. 7 Synchronous and asynchronous communication

Synchronous	Asynchronous
REST over HTTP or gRPC; caller waits for the response	Messages/events sent through a broker; caller continues immediately
Simple, easy to debug	Loose coupling, better resilience and throughput
Failure of callee affects caller	Messages are stored until the consumer is ready
Use for queries needing an immediate answer	Use for notifications, long-running work, event propagation

Messaging styles: point-to-point queue (one consumer processes each message) and publish–subscribe topic (every subscriber gets a copy). Because messages may be delivered more than once, consumers should be idempotent.

2.5 Data Management

- **Database per service:** each service owns its data and exposes it only through its API; no cross-service joins.
- **Eventual consistency:** data in different services becomes consistent after events are processed, instead of a single ACID transaction.

- **Saga pattern:** a business transaction is split into local transactions; if one fails, compensating actions undo the earlier steps (e.g. cancel the order if payment fails). Implemented by choreography (services react to events) or orchestration (a coordinator directs the steps).
- **CQRS and event sourcing:** separate read and write models; store changes as a sequence of events.

2.6 Important Microservice Design Patterns

Pattern	Purpose
API Gateway	Single entry point and cross-cutting concerns for all clients
Service Discovery	Find service instances dynamically (client-side or server-side)
Database per Service	Loose coupling of data
Circuit Breaker	Stop calling a failing service and fail fast
Retry, Timeout, Bulkhead	Handle transient failures and isolate resource pools
Saga	Manage distributed transactions
Backend for Frontend (BFF)	Separate gateway tailored to each client (web, mobile)
Sidecar / Service Mesh	Helper proxy next to each service for security, routing, telemetry (Istio, Linkerd)
Strangler Fig	Gradually replace a monolith by moving features to new services
Externalised Configuration	Keep settings outside the code

Circuit Breaker

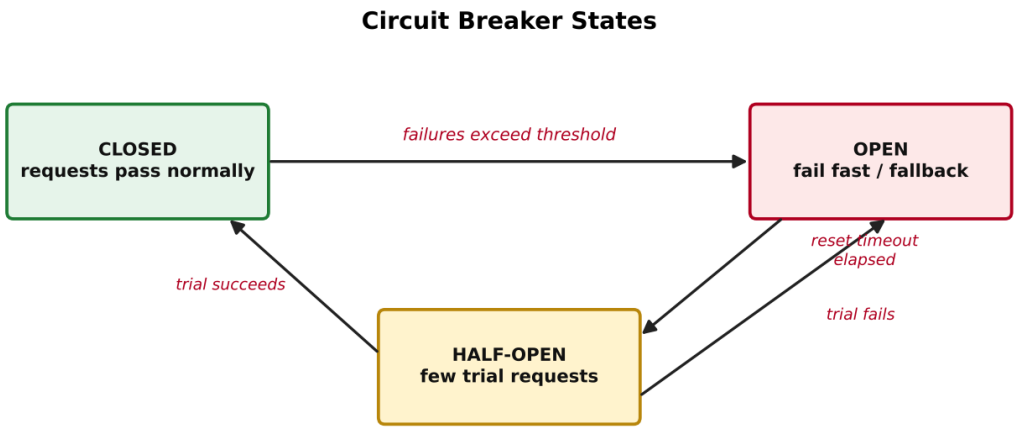


Fig. 8 Circuit breaker states

When calls to a service keep failing, continuing to call it wastes resources and makes the caller slow. The circuit breaker monitors failures: in the CLOSED state calls pass normally; when failures exceed a threshold it moves to OPEN and fails immediately (or returns a fallback); after a reset timeout it becomes HALF-OPEN and lets a few trial requests through – success closes the circuit, failure opens it again.

```
const CircuitBreaker = require("opossum");

const breaker = new CircuitBreaker(callPaymentService, {
  timeout: 3000,                    // a call slower than 3 s counts as
failure
  errorThresholdPercentage: 50,     // open when 50% of calls fail
  resetTimeout: 10000,              // try again after 10 s (half-open)
});
breaker.fallback(() => ({ status: "PENDING" }));
const result = await breaker.fire(order);
```

2.7 Example – E-Commerce Application

Service	Responsibility	Sample endpoints	Database
User Service	Registration, login, profiles	POST /api/users, POST /api/login	MySQL
Product Service	Catalogue and stock	GET /api/products, GET /api/products/{id}	MongoDB
Order Service	Create and track orders	POST /api/orders, GET /api/orders/{id}	PostgreSQL
Payment Service	Process payments	POST /api/payments	PostgreSQL
Notification Service	E-mail / SMS on events	Consumes order.created, payment.success	Queue only

API Gateway using Express

```
const express = require("express");
const { createProxyMiddleware } = require("http-proxy-middleware");
const app = express();

app.use(createProxyMiddleware({ target: "http://localhost:4001",
changeOrigin: true,
                                pathFilter: "/api/products" }));   // Product
Service
app.use(createProxyMiddleware({ target: "http://localhost:4002",
changeOrigin: true,
                                pathFilter: "/api/orders" }));     // Order
Service

app.listen(8080, () => console.log("Gateway on 8080"));
```

Order Service calling Product Service

```
app.post("/api/orders", express.json(), async (req, res) => {
  try {
    const { productId, qty } = req.body;
    const r = await
fetch(`${process.env.PRODUCT_URL}/api/products/${productId}`);
    if (!r.ok) return res.status(404).json({ error: "Product not found" });
    const product = await r.json();
    if (product.stock < qty) return res.status(409).json({ error:
"Insufficient stock" });
    const order = await Order.create({ productId, qty, total: product.price *
qty });
```

```
    res.status(201).json(order);
  } catch (e) {
    res.status(503).json({ error: "Product service unavailable" });
  }
});
```

2.8 Deployment of Microservices

Each service is packaged as a container image with Docker, so it runs identically on every machine. An orchestrator such as Kubernetes starts several instances, load-balances them, scales them automatically and restarts failed ones. A CI/CD pipeline builds, tests and deploys each service independently.

```
# Dockerfile for one Node.js service
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

```
# docker-compose.yml (local development)
services:
  gateway:
    build: ./gateway
    ports: ["8080:8080"]
    depends_on: [product-service, order-service]
  product-service:
    build: ./product-service
    environment:
      - MONGO_URI=mongodb://product-db:27017/products
  order-service:
    build: ./order-service
    environment:
      - PRODUCT_URL=http://product-service:3000
  product-db:
    image: mongo:7
```

2.9 Observability and Security

Area	Techniques
Logging	Centralised logs (ELK / Loki) with a correlation id passed through all calls
Metrics	Prometheus + Grafana: request rate, errors, latency
Distributed tracing	OpenTelemetry, Jaeger, Zipkin to follow one request across services
Health checks	/health endpoints used by Kubernetes and load balancers
Authentication	Gateway validates the JWT and forwards identity to services
Service-to-service security	mutual TLS, service mesh, short-lived tokens
Secrets	Vault / Kubernetes Secrets, never in the image or Git

2.10 Advantages, Disadvantages and When to Use

Advantages	Disadvantages
Independent deployment and faster releases	Distributed-system complexity (network failures, latency)
Scale only the services that need it	Harder testing and debugging across services
Fault isolation and better resilience	Data consistency is eventual, not immediate
Technology flexibility; small focused teams	Operational overhead: DevOps, monitoring, many deployments
Easier to understand each small codebase	Higher infrastructure cost for small applications

Use microservices for large, evolving applications with several teams and different scaling needs. For a small application or a new product with unclear boundaries, a well-structured monolith is usually the better starting point.

2.11 Microservices vs SOA

SOA	Microservices
Enterprise-wide, coarse-grained services	Fine-grained, single-purpose services
Often shared database and heavy middleware (ESB)	Database per service, lightweight communication
SOAP / XML common	REST / JSON / messaging common
Services may share components	Services are independent and share nothing

3. API Documentation using Swagger / OpenAPI

3.1 Need for API Documentation

An API is only useful if developers can understand how to call it. Good documentation lists the available endpoints, HTTP methods, parameters, request and response formats, authentication method, status codes and examples. Manually written documents quickly become outdated; a machine-readable specification solves this because the same file can generate documentation, client libraries and tests.

3.2 OpenAPI Specification and Swagger

- **OpenAPI Specification (OAS):** a standard, language-independent format (YAML or JSON) for describing REST APIs. It is maintained by the OpenAPI Initiative under the Linux Foundation.
- **Swagger:** the original name of the specification (Swagger 2.0). The specification was donated in 2015 and renamed OpenAPI; today 'Swagger' refers to the tool set from SmartBear built around the OpenAPI Specification.

- **Versions:** Swagger 2.0, OpenAPI 3.0.x and 3.1.x are the ones met most often (3.1 aligns the schema language with JSON Schema). Newer minor versions continue to appear, so check the current release when starting a project.

Swagger / OpenAPI tools

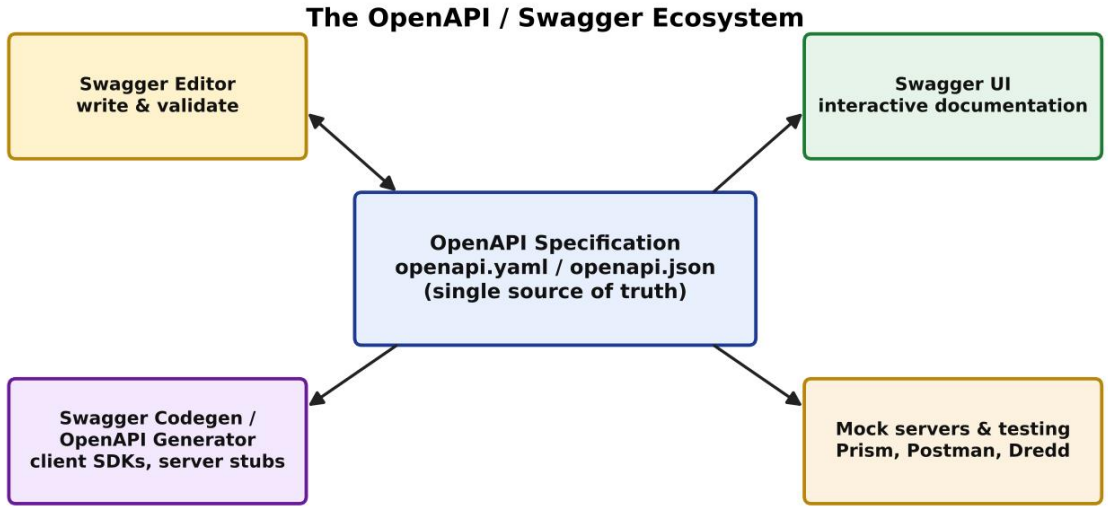


Fig. 9 The OpenAPI / Swagger ecosystem

Tool	Purpose
Swagger Editor	Browser-based editor that validates the YAML and shows a live preview
Swagger UI	Generates interactive documentation; developers can call the API using the 'Try it out' button
Swagger Codegen / OpenAPI Generator	Generates client SDKs and server stubs in many languages from the specification
SwaggerHub	Cloud platform for designing, sharing and collaborating on API definitions
Mock / testing tools	Prism, Postman, Dredd – run a fake server or test the API against the contract
Linters	Spectral – checks the specification against style rules

3.3 Structure of an OpenAPI Document

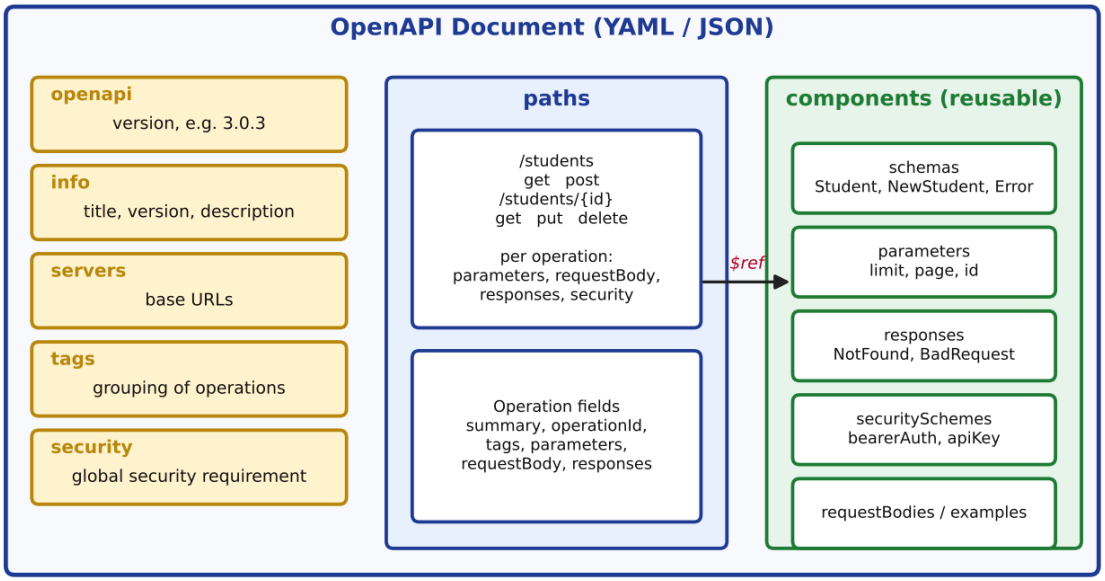


Fig. 10 Structure of an OpenAPI document

Field	Description
openapi	Version of the OpenAPI Specification used (e.g. 3.0.3)
info	Metadata: title, version of the API, description, contact, licence
servers	Base URLs of the API (production, staging, localhost)
tags	Names used to group operations in the documentation
paths	The endpoints. Each path contains operations (get, post, put, patch, delete)
components	Reusable definitions: schemas, parameters, responses, requestBodies, securitySchemes, examples
security	Security requirements applied to all operations (can be overridden per operation)

Parts of an operation

Element	Meaning
summary / description	Short and detailed explanation of what the operation does
operationId	Unique name, used by code generators as the method name
tags	Group in the UI, e.g. Students
parameters	Inputs: in = path, query, header or cookie; name, required, schema
requestBody	Body sent by the client with content type (application/json) and schema
responses	One entry per status code (200, 201, 404 ...) with description and schema
security	Authentication needed (e.g. bearerAuth)

Element	Meaning
$ref	Reference to a reusable definition, e.g. '#/components/schemas/Student'

3.4 Example – OpenAPI Definition of a Student API (YAML)

```yaml
openapi: 3.0.3
info:
  title: Student API
  description: Simple CRUD API for managing students
  version: 1.0.0
servers:
  - url: http://localhost:3000/api/v1
tags:
  - name: Students
    description: Operations on students

paths:
  /students:
    get:
      tags: [Students]
      summary: List students
      operationId: listStudents
      parameters:
        - name: dept
          in: query
          schema: { type: string }
          description: Filter by department
        - name: limit
          in: query
          schema: { type: integer, default: 10, maximum: 100 }
      responses:
        '200':
          description: List of students
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Student'
    post:
      tags: [Students]
      summary: Create a student
      operationId: createStudent
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/NewStudent'
      responses:
        '201':
          description: Student created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Student'
        '400':
```

```
          $ref: '#/components/responses/BadRequest'

  /students/{id}:
    parameters:
      - name: id
        in: path
        required: true
        schema: { type: integer }
    get:
      tags: [Students]
      summary: Get a student by id
      operationId: getStudent
      responses:
        '200':
          description: The student
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Student'
        '404':
          $ref: '#/components/responses/NotFound'
    put:
      tags: [Students]
      summary: Replace a student
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/NewStudent'
      responses:
        '200': { description: Updated }
        '404':
          $ref: '#/components/responses/NotFound'
    delete:
      tags: [Students]
      summary: Delete a student
      security:
        - bearerAuth: []
      responses:
        '204': { description: Deleted }
        '404':
          $ref: '#/components/responses/NotFound'

components:
  schemas:
    NewStudent:
      type: object
      required: [name, dept]
      properties:
        name: { type: string, example: Anu }
        dept: { type: string, enum: [CSE, IT, ECE], example: IT }
        year: { type: integer, minimum: 1, maximum: 4, example: 3 }
    Student:
      allOf:
        - $ref: '#/components/schemas/NewStudent'
        - type: object
          properties:
            id: { type: integer, example: 7 }
    Error:
```

```
      type: object
      properties:
        error: { type: string, example: Student not found }
  responses:
    NotFound:
      description: Resource not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
    BadRequest:
      description: Invalid input
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Observe that the repeated Student, NewStudent, Error and response definitions are written once under components and re-used with $ref. This keeps the document short and consistent. In YAML, indentation (spaces, not tabs) defines the structure.

3.5 Design-First vs Code-First

Design-First (contract-first)	Code-First
The OpenAPI file is written first, reviewed and agreed	The API code is written first; the spec is generated from annotations
Front-end and back-end teams can work in parallel using mocks	Faster for small projects and quick prototypes
Spec is the single source of truth and drives code generation	Documentation may drift if annotations are not maintained
Preferred for public APIs and microservices contracts	Common with frameworks such as Spring (springdoc) or Express (swagger-jsdoc)

3.6 Integrating Swagger UI with Express

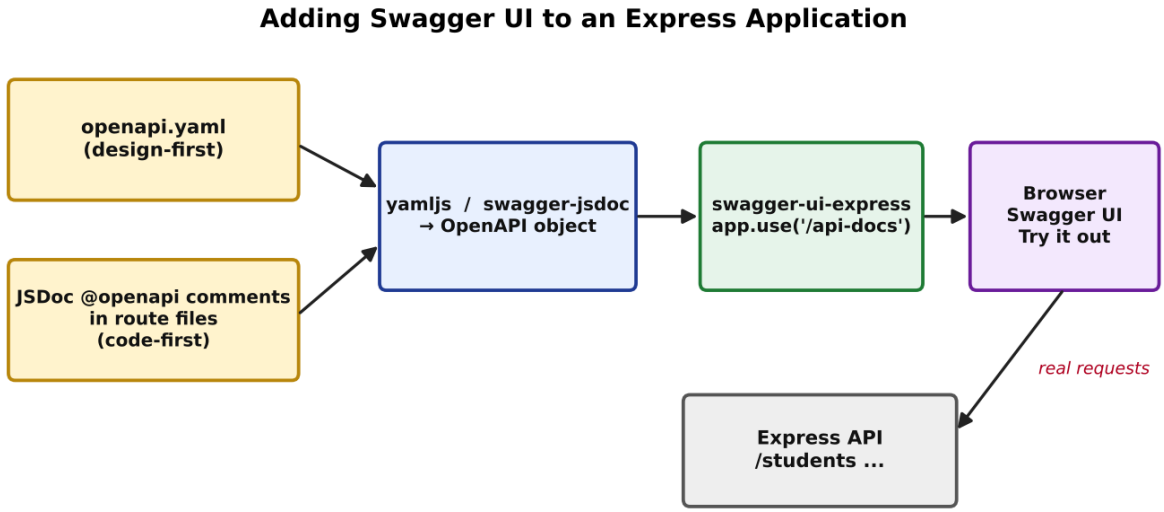


Fig. 11 Adding Swagger UI to an Express application

```
npm install swagger-ui-express yamljs swagger-jsdoc
```

(a) Design-first: serve an existing openapi.yaml

```
const express = require("express");
const swaggerUi = require("swagger-ui-express");
const YAML = require("yamljs");

const app = express();
const spec = YAML.load("./openapi.yaml");

app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(spec));
app.listen(3000);       // open http://localhost:3000/api-docs
```

(b) Code-first: generate the spec from JSDoc comments

```
const swaggerJsdoc = require("swagger-jsdoc");

const options = {
  definition: {
    openapi: "3.0.3",
    info: { title: "Student API", version: "1.0.0" },
    servers: [{ url: "http://localhost:3000/api/v1" }],
  },
  apis: ["./routes/*.js"],          // files containing @openapi comments
};
app.use("/api-docs", swaggerUi.serve,
swaggerUi.setup(swaggerJsdoc(options)));
```

```
// routes/studentRoutes.js
/**
 * @openapi
 * /students/{id}:
 *   get:
 *     tags: [Students]
 *     summary: Get a student by id
 *     parameters:
```

```
 *       - in: path
 *         name: id
 *         required: true
 *         schema: { type: integer }
 *     responses:
 *       200:
 *         description: The student
 *       404:
 *         description: Not found
 */
router.get("/:id", ctrl.getOne);
```

Using Swagger UI

- Operations are grouped by tag; each can be expanded to see parameters, request body schema, responses and examples.
- 'Try it out' fills in sample values, sends a real request to the server and shows the response, status code and headers.
- The 'Authorize' button stores a bearer token so that secured operations can be tested.
- The 'Schemas' section lists all models from components.
- The raw specification can be downloaded and imported into Postman or a code generator.

3.7 Generating Client Code and Mock Servers

```
# generate a TypeScript client from the spec
npx @openapitools/openapi-generator-cli generate -i openapi.yaml -g
typescript-axios -o ./client

# validate / lint the spec
npx @stoplight/spectral-cli lint openapi.yaml

# run a mock server that returns the example responses
npx @stoplight/prism-cli mock openapi.yaml
```

3.8 Other Documentation Tools and Formats

Tool / format	Remarks
Postman collections	Documentation and testing generated from saved requests; can import OpenAPI
ReDoc	Clean three-panel documentation rendered from an OpenAPI file
Stoplight, Insomnia	Design and documentation platforms supporting OpenAPI
RAML, API Blueprint	Older alternative description languages
AsyncAPI	Specification for event-driven APIs (Kafka, MQTT, WebSocket) – like OpenAPI for messaging
GraphQL schema / introspection	GraphQL APIs document themselves through their schema

3.9 Best Practices for API Documentation

- Describe every endpoint, parameter and response, including error responses (400, 401, 404, 500).
- Provide realistic examples for requests and responses.
- Reuse definitions through components and $ref to avoid duplication.
- Document authentication clearly and show how to obtain a token.
- Version the specification together with the API and keep a change log.
- Validate and lint the specification in the CI pipeline so the documentation never drifts from the code.
- Use contract testing to check that the real API matches the specification.

OpenAPI in a microservices system

Each microservice publishes its own OpenAPI file, which acts as its contract with the other services and front-end teams. The API gateway or a developer portal can collect these files and present one combined catalogue, and consumer teams can generate clients or mock servers from them without waiting for the service to be finished.