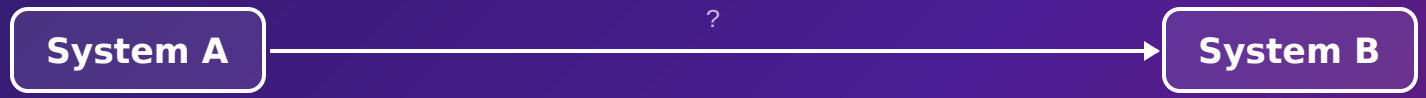
API INTEGRATION KNOWLEDGE SERIES · PART 1 OF 4

How systems really talk to each other

The API & integration landscape — a working map for analysts.



Marko Hartikainen 9 July 2026

BEFORE WE START · 2 MINUTES

Quick self-assessment — rate 1 to 5

- I can explain the difference between an API, an integration, an interface, and an event.
- I can compare common integration styles and justify a technology choice — including recognizing when an API is the wrong answer.
- I can list the key functional and non-functional requirements for an API integration and notice what is missing.
- I can explain the main REST API security controls in simple language and recognize when to escalate.
- I can read a basic OpenAPI document, propose test scenarios from it, and use an LLM safely to support this work.

Anonymous · the same five statements return at the end of session 4 — we measure the series, not you.

ORIENTATION

Every project has an integration in it

- Most analyst pain lives at system boundaries — not inside systems.
- The vocabulary gap is expensive: it costs us influence in architecture discussions and precision in requirements.
- Nobody here needs to become a developer. We need to **ask better questions, earlier**.

THE SERIES

Four sessions, one growing checklist

1 · today

Understand & choose

The landscape, the trade-offs, and when an API is the wrong answer.

2 · TBA

Analyse & specify

Requirements, design contribution, and who to contact, consult, or inform.

3 · TBA

Secure

REST security layer by layer, in a banking context. You spot the weaknesses.

4 · TBA

Operate & work smarter

Documentation, testing, lifecycle, risks — and using LLMs safely.

Each session ends by adding its sections to a shared **API Integration Analyst Checklist**. It stays with you after the series.

TODAY

By the end of this session

- You can place the common integration styles on one map — current and legacy.
- You can apply a five-question framework to route a business need to a technology family.
- You can say "*this should not be an API*" — with reasons, and with alternatives.

`Vocabulary → the axis → the landscape → choosing → when not to → your turn.`

API ≠ integration

- **Interface** — any agreed boundary: a UI, a file format, an API.
- **API** (Application Programming Interface) — a machine-usable interface a system exposes: software talking to software, no human in the loop.
- **Integration** — everything that makes two systems actually work together: the API *or* file *or* queue, plus mapping, schedule, error handling, ownership.

An API is a door. An integration is the whole delivery route — the truck, the schedule, and what happens when the truck breaks down. "*There's an API for that*" is where analysis starts, not where it ends.

VOCABULARY

Command vs event

`command · "do this"`

A request aimed at one system, expecting an outcome: `CancelPolicy(123)`. The caller cares about the answer.

`event · "this happened"`

A statement of fact, published to whoever subscribes: `PolicyCancelled(123)`. No response expected — the publisher doesn't know who listens.

A **service** is a component offering capability — usually through APIs, sometimes by publishing events.

An API is a product, not a project

- A real API is a **lifecycle-managed asset**: it has an owner, a version, known consumers, and — one day — a deprecation plan. Not just an endpoint that happens to exist.
- The analyst questions that follow: who owns it? which version do we bind to? how are changes announced? who else consumes it — and what happens to them when it changes?
- The inverse also holds: "*we need an API*" is sometimes the wrong starting point. First ask what the business needs to move or know — today's framework does exactly that.

> **An endpoint without an owner is not an API — it is a future incident.** Lifecycle & governance get their own slot in session 4.

Synchronous vs asynchronous

Synchronous — the caller waits

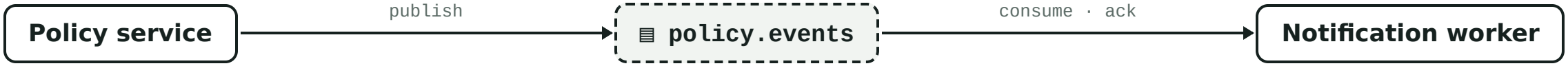
- Request → response, one conversation.
- Simple mental model, easy to test.
- Runtime dependency: if B is down or slow, A hurts *right now*.

Asynchronous — hand it off

- Fire-and-forget, queues, publish/subscribe.
- Resilient and decoupled; absorbs bursts.
- Harder to analyse, trace, and test — the work happens *later, elsewhere*.

The single most important question in any integration discussion: does the caller need the answer to proceed?

A queue between two systems



worker offline? messages wait in the queue — nobody crashes · failed message retries · poison messages parked in a dead-letter queue

- **Store-and-forward:** the queue holds every message until the consumer is ready — bursts are absorbed, nothing is lost in transit.
- Failure is isolated: the publisher never notices a slow consumer.
- New analyst questions appear: ordering? duplicates? how long may a message wait?

Real-time · near-real-time · batch

milliseconds · sync

Card authorization

The customer is standing at the terminal. The answer must come now.

seconds–minutes · events

Fraud alerting

Fast matters, but nobody is blocked waiting. Event-driven fits.

nightly · files

Statements & interest runs

Scheduled, bulk, auditable. Batch is the honest answer.

"Real-time" in a requirement is a cost decision. Always ask: how real, measured in what, and what breaks if it's a minute late?

The map for the next 25 minutes

request / response

- REST
- GraphQL
- gRPC
- SOAP*

push

- webhooks
- WebSockets

messaging

- queues
- pub/sub
- event streams

bulk

- files · SFTP
- EDI*
- ETL / ELT

infrastructure

- API gateway
- iPaaS
- ESB*

* legacy-leaning, still very much alive · every style gets one to two minutes — depth on request via the parking lot

REST — the default

Representational State Transfer · in practice: JSON over HTTP(S)

- Resources addressed by URLs, standard HTTP verbs (GET, POST, PUT, DELETE), JSON payloads, stateless calls.
- **Reach for it:** partner-facing and general request/response — tooling everywhere, easiest onboarding.
- **Watch:** not built for high-volume streams; versioning needs discipline.

HTTP · request / response

```
$ curl -s
https://api.cybernotes.it/mtpl/v1/coverage-options

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

{ "options": [
    { "code": "MTPL-STD",
      "name": "Mandatory motor liability",
      "term": "P1Y" } ] }
```

Simulated call — this fictional insurer API goes live later in the series.

GraphQL — the client decides the shape

- One endpoint; the client asks for exactly the fields it needs, across sources.
- **Reach for it:** flexible, read-heavy aggregation — mobile apps, backend-for-frontend layers.
- **Watch:** server-side complexity, field-level authorization, caching — the flexibility is paid for somewhere.

> Analyst question: is the need really *shape flexibility* — or just three missing fields in an existing REST API?

gRPC — fast internal plumbing

gRPC Remote Procedure Calls · call a function on another system as if it were local

- HTTP/2 + Protocol Buffers: binary, compact, contract-first (`.proto` files), supports streaming.
- **Reach for it:** high-performance service-to-service calls inside your own estate.
- **Watch:** not browser- or partner-friendly by default; payloads aren't human-readable on the wire.

Webhooks

- The provider POSTs to *your* URL when something happens.
- **Reach for it:** callbacks and notifications from partners and SaaS.
- **Watch:** your side must verify signatures (HMAC — a hash-based message authentication code; live demo in session 3), handle retries, and cope with the same event arriving twice.

WebSockets

- A persistent, two-way channel between client and server.
- **Reach for it:** live UIs — prices, chat, dashboards.
- **Watch:** stateful connections are an operations burden; not for system-to-system data exchange.

Message queues — decoupling & load leveling

- Reliable handoff: each message is acknowledged after successful processing; failures retry; poison messages park in a dead-letter queue.
- Competing consumers: scale workers up or down without touching the publisher.
- **Reach for it:** asynchronous commands, burst absorption, "must not be lost, need not be instant". e.g. RabbitMQ and similar brokers

One message → exactly one worker, then it's gone. But what if *several* systems need to know? Next slide.

Publish / subscribe — tell everyone at once

- An event is published once; the broker delivers a copy to **every subscriber**. The publisher never knows who is listening.
- **Event notification** — the lightest form: `PolicyCancelled(123)` announces the fact; subscribers react, or call back for details if they need more.
- Adding a subscriber later costs the publisher **nothing** — the decoupling superpower a point-to-point queue alone doesn't give you.
- **Watch:** silent subscribers — if one listener fails quietly, who notices? Each subscription is its own mini-integration with its own monitoring and error handling.

Subscribers only hear what happens while they are subscribed. Need history as well? Next slide.

Event streaming — the replayable log

- An append-only log of events; consumers keep their own position and can **replay history**.
- Many independent readers of the same events — analytics, fraud, notifications, each at its own pace.
- **Reach for it:** high volume, many consumers, event history matters. `e.g. Kafka and similar platforms`

> **Queue:** message consumed → gone. **Stream:** events remain → readers move. If someone says "Kafka" where you expected "queue", this is the difference they mean.

API gateway

- One front door for many APIs.
- Gives you authentication, rate limits, quotas, routing and logging **"for free"**.
- Analyst impact: never write requirements the gateway already fulfils — reference them instead.

iPaaS (and its ancestor, the ESB)

- iPaaS (Integration Platform as a Service): cloud platforms with pre-built connectors to common SaaS and enterprise systems.
- **Reach for it:** standard system pairs — configure, don't build.
- The ESB (Enterprise Service Bus) is the older, centralized on-premise version — you will still meet it.

You will meet these in the wild

Recognition level: name it, place it on the map, know its implications, know who to call.

No nostalgia, no disdain — these move real money every night.

STILL ALIVE

SOAP & XML over HTTP

- SOAP (originally "Simple Object Access Protocol"): XML envelopes with formal **WSDL** (Web Services Description Language) contracts and strict typing; WS-Security for enterprise-grade needs.
- "XML over HTTP" is the informal cousin — the same XML payloads POSTed to a URL, minus the envelope and the formal contract.
- Where: older vendor systems, government interfaces, core enterprise platforms.
- Implication: contract-first is not a new idea — SOAP enforced it strictly. Expect rigid change processes and, usually, a vendor on the other end.

Files still move the money

naming convention · duplicate-file handling · missing-file alerting · reconciliation of counts and sums

- Payment files, reconciliations, nightly loads over SFTP (file transfer through an encrypted SSH channel): simple, auditable, schedule-bound.
- **EDI** (Electronic Data Interchange) — standardized business documents (EDIFACT and friends) — lives in the same family.
- File integrations deserve full requirements rigor too — session 2 treats them as first-class citizens.

discouraged, not extinct

Direct DB integration

Reading or writing another system's tables (or calling its stored procedures) couples you to its schema and bypasses its business logic and audit. You will still find it.

right tool, right job

ETL / ELT

Extract–Transform–Load (or load first, transform in the warehouse). Bulk data movement into analytics and reporting estates — correct for data pipelines, wrong for transactional process integration.

it runs the world quietly

Mainframe patterns

MQ bridges, file drops, terminal emulation. Recognize the words; escalate the details.

Seven axes that decide

Coupling

How tightly do the two ends depend on each other?

Speed

Latency the business actually needs — not wants.

Reliability

What may be lost, duplicated, or late?

Ownership

Who controls each end — and the contract between them?

Complexity

Build and run effort, including testing.

Security

Data sensitivity, exposure surface, controls required.

Cost

Licences, platforms, people — over the whole lifetime.

Full matrix

In the appendix and the analyst checklist — reference material, not memory work.

Five questions route you — watch them work

Scenario: the policy system hands renewal letters to the document service — thousands in a nightly burst, none may be lost, nobody sits waiting.

— **1 · Does the caller need the answer to proceed?**
No — nobody is waiting → asynchronous.

— **2 · Data movement or process interaction?**
A command — "produce this letter" — not a data copy.

— **3 · What volume, and how bursty?**
Thousands at 02:00 → we need a buffer that absorbs peaks.

— **4 · What happens when the other side is down?**
Work waits and resumes — one letter, one job, done once. No history to replay.

— **5 · Who controls both ends?**
Both ends are ours → no partner constraints, we choose freely.

~~REST~~ ~~GraphQL~~ ~~gRPC~~ ~~webhook~~ **queue**

~~event stream~~ ~~file · SFTP~~ ~~ETL / ELT~~

One left: the message queue — reliable handoff, burst absorption, dead-letter safety net. e.g. RabbitMQ

these five go into the analyst checklist — and your homework API gets the same treatment in session 2

Reach for it when…

REST	default request/response; partners; broadest tooling
GraphQL	client-shaped reads across several sources
gRPC	fast internal service-to-service calls
SOAP	the counterpart requires it — formal XML contracts, older vendor & government systems
webhook	a provider must push events to you
queue	reliable async handoff; burst absorption
event stream	high volume, many readers, replayable history
file · SFTP	bulk on a schedule; audit-friendly; legacy counterpart
ETL · ELT	data movement into analytics and reporting
manual · RPA	temporary bridge only — time-boxed, with a written exit plan

Sometimes the right answer is no API

The framework's most valuable output is permission to say no — with reasons, and with an alternative in hand.

Ten recurring cases follow. Each one has appeared in a real project near you.

CASES 1–5 · SITUATION → ALTERNATIVE

WHEN…	REACH FOR…
Batch is enough for the business	files · ETL
Real-time adds no business value	scheduled exchange
A direct API would create tight runtime coupling	events · messaging
The source API is unstable or unsupported	file export · wait for a stable contract
Event- or message-based fits the flow better	publish / subscribe

WHEN…	REACH FOR…
An existing connector or iPaaS covers the pair	`configure, don't build`
ETL / ELT is the correct pattern (data, not process)	`data pipeline`
File exchange is simply sufficient	`SFTP + reconciliation`
Manual / RPA (Robotic Process Automation) workaround bridges a gap	`time-boxed, with a written exit plan`
Data ownership is not clear enough to integrate safely	`stop — fix ownership first`

YOUR TURN · 10 MINUTES

Three scenarios — think along

- In a room: **say it out loud**. Working through this alone: **pause and commit** to an answer before you hear mine.
- For each scenario: propose an integration style *and* answer — could the right answer be no API?
- Expect follow-up questions — some of them tricky on purpose. A confident wrong answer moves us further than silence.

Route with the five questions. Disagreement — with me or each other — is a feature.

SCENARIO 1 OF 3

"A customer profile update must be visible immediately in a downstream system."

— How immediate is *immediately* — and who says so?

— Who owns the profile — which system is the system of record?

— What should happen when the downstream system is unavailable?

SCENARIO 2 OF 3

"A transaction notification must be delivered reliably — immediate processing is not mandatory."

- Reliability versus latency — which one is the hard requirement here?
- What happens to a notification that fails processing three times?
- Does the order of notifications matter?

SCENARIO 3 OF 3

"The legacy system can only export files once per night."

- Is nightly actually enough for the business need — has anyone asked?
- Duplicate file, missing file, half-written file: who notices, and how?
- Who reconciles counts and sums — and against what?

WRAP · YOUR TAKE-HOME FROM THIS SERIES

What you'll be able to do — and what you just need to recognize

do yourself · the hands-on goal of this series

- Look at a real HTTP request and response and explain what happened: which operation was called, with what data, and whether it succeeded — like the example on the REST slide.
- Open an API's OpenAPI document and find what the API offers: the operations, the required fields, the example responses.
- Steer an integration discussion using the five routing questions — starting with "*does the caller need the answer to proceed?*"
- Write solid requirements for an integration — errors, volumes, ownership included. That's the craft of session 2.

recognize · enough to react correctly

- Hear *SOAP, EDI, Kafka* or *gRPC* in a meeting and know where it sits on today's map — and what it means for your project.
- Know which concerns the API gateway already covers — authentication, rate limits, logging — so you reference them instead of re-specifying them.
- Sense when a topic goes beyond analyst territory — and pull in architects or security early, with a sharp question in hand.

Nobody here is becoming a developer. The left card is the whole ambition — and every item on it is learnable.

BEFORE YOU GO · ONE PAGE, ONE API, ONE DATE

Today moves to your checklist — you bring one API

right now

Everything from today, one page

The map, the five questions, the ten "no API" cases, the matrix, the scenarios — plus the analyst checklist that grows each session.

link → on the training page

try at home · 15 min

Start from an API you already know

Pick one integration or API you've had contact with in your organisation. Place it on today's map. Then write down: **which questions does it answer** for its callers — and one question about it *you* can't answer yet.

for session 2

Keep it in mind

Start part 2 with that API in the back of your head. When we walk through the requirements checklist, test every point against it — that's when what an API **really requires** clicks.

slides → cybernotes.it/trainings/api-integration/1-integration-landscape · PDF on the same page · part 2: requirements & design — published

APPENDIX · COMPARISON MATRIX — REFERENCE, NOT MEMORY WORK

STYLE	COUPLING	LATENCY	FAILURE ISOLATION	COMPLEXITY	SECURITY SURFACE	TYPICAL COUNTERPART	COST
REST	med	low	low	low	med	partner or internal	low
GraphQL	med	low	low	med–high	med–high	own frontend / BFF	med
gRPC	med	very low	low	med	med	internal services	med
SOAP	high	low	low	med–high	med	vendor or government	med
webhook	low	low–med	med	med	high	provider → you	low
queue	low	async	high	med	med	internal systems	med
event stream	low	async	high	high	med–high	many internal readers	med–high
file · SFTP	low	schedule	high	low	low–med	partner or legacy	low
ETL · ELT	low	schedule	high	med	low–med	analytics estate	med
direct DB	very high	low	low	low — deceptively	high	legacy · internal	low now · high later

failure isolation = how insulated you are when the other side is down · security surface = exposure & controls required · coarse ratings on purpose — context beats matrices

APPENDIX · JARGON DECODER — TODAY'S TERMS IN ONE PLACE

API	Application Programming Interface — a machine-usable interface a system exposes
REST	Representational State Transfer — resource-style APIs over HTTP, usually JSON
HTTP·HTTPS	the web's transfer protocol · S = encrypted with TLS (Transport Layer Security)
JSON	JavaScript Object Notation — the default text format for payloads
OpenAPI	the standard machine-readable description of a REST API contract
gRPC	gRPC Remote Procedure Calls — fast binary calls over HTTP/2, .proto contracts
SOAP	XML protocol with formal contracts (originally Simple Object Access Protocol)
WSDL	Web Services Description Language — SOAP's contract file
HMAC	hash-based message authentication code — proves who sent a message, unaltered

SFTP	SSH File Transfer Protocol — file exchange over an encrypted channel
EDI	Electronic Data Interchange — standardized business documents (e.g. EDIFACT)
ESB	Enterprise Service Bus — centralized on-premise integration hub (legacy)
iPaaS	Integration Platform as a Service — cloud connector platform
ETL·ELT	Extract, Transform, Load — bulk data movement into analytics estates
RPA	Robotic Process Automation — software robots driving user interfaces
DLQ	dead-letter queue — parking lot for messages that repeatedly fail processing
BFF	Backend for Frontend — an API layer shaped for one specific client
MTPL	Motor Third-Party Liability — the fictional demo API's insurance domain

not presented live — lives in the PDF as a lookup · security terms (OAuth2, JWT, mTLS …) get their own decoder in session 3