The Lazy Dev's Guide to Beautiful APIs

🚀 *Pragmatic APIs That Scale*

Apipana.io®
CODING SOCIETY

#pulpocon - 2025
#gastrotech

Who I am?

Jesús María Villar Vázquez

Just a Developer working for @apipana.io #wearehiring

20 years of experience

I love Testing, eXtreme Programming and Coding

geeksusma

APIs You Can Be Proud Of 🚀🏆

🛠️ Build APIs that make developers *smile*

🎯 Focus on clarity, not complexity

👌 Quality without the over-engineering headache

What This Talk Is About 🤔 💡

🎯 Pragmatic API design principles

🏆 Why REST Level 2 hits the sweet spot

🚫 Avoid over-engineering traps

🏎️ A little bit of CQRS (only a little, I promise)

🤝 How governance keeps teams in sync

The Cheat Sheet

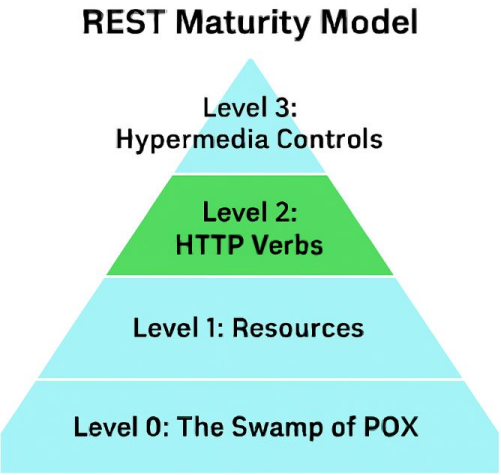
Practical tips to build APIs you won’t HATE later

Real Talk: What Do We Actually Need? 🤦💥

✅ REST Level 2 = “Good Enough”

🔑 Use resources, HTTP methods & status codes right

🚫 Skip “pure REST” fan club debates



Naming Things: Leave a Good Legacy 🏷️📚

Use **nouns**, not verbs
❌ `/createUser` → ✅ `POST /users`

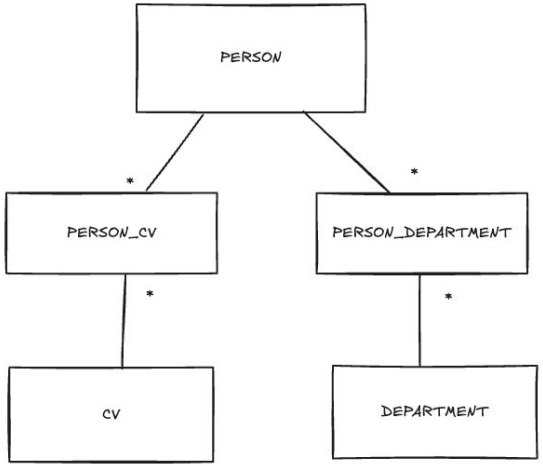
Plural resources for collections
✅ `/users`, `/products`, `/orders`

Be boring, consistent, and predictable
📌 Every endpoint is part of your interface contract

Naming Things: Proper Abstraction Layer 🏷️📚

The common pitfall, mapping your endpoints to your persistence layer
❌ `/person` ❌ `/person-department` ❌ `/person?isEmployee=true`

Take advantage of using the proper abstraction layer
✅ `/employees`, `/candidates`, `/employees/{id}/departments`, `/candidates/{id}/cvs` `/cvs`



Naming Things: Obfuscate Id’s 🛡️🥷

An Id must be something hard to guess. So it is highly encouraged to avoid sequences of numbers or string patterns that are easy to guess. Protect your data!

❌ `/persons/1` ❌ `/persons/2` ❌ `/persons/jesusmvillarvazquez`

Obfuscate the ID’s using UUID

✅ `/persons/e96b5514-6e8d-45fa-85f6-eda8580b7832`

HTTP Methods: Use Them Like You Mean It 🕹️🎯

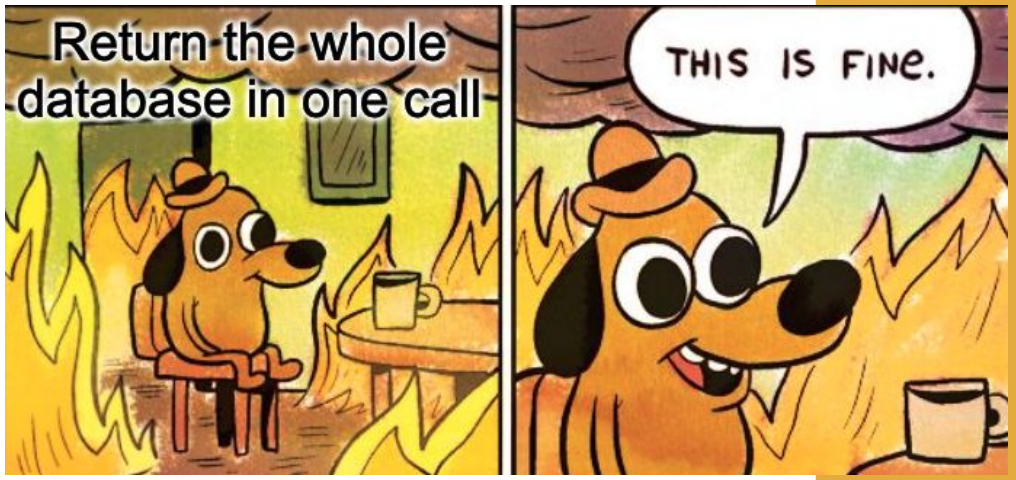
Method	Action
GET	Read data
POST	Create
PUT	Replace
PATCH	Update partially / Commands
DELETE	Remove

- GET = Safe
- POST = Non-idempotent
- PUT = Idempotent
- PATCH = Partial
- DELETE = Final

📌 Method misuse = unexpected behavior for consumers

Pagination: Do It, Always 📚➡️

- 🛑 Never dump huge data all at once
- 📄 Use page or cursor-based pagination
- 📊 Always include metadata (next, total, etc.)



Error Handling: Use Problem JSON 🚨🧩

📜 RFC 7807: `application/problem+json`

🧩 Structured, consistent errors with `type`, `title`, `status`, `detail`

🤖 Machine-friendly & extendable

```
{
  "type": "https://example.com/probs/invalid-email",
  "title": "Invalid email address",
  "status": 400,
  "detail": "The email 'abc' is invalid",
  "instance": "/signup/abc"
}
```

Response Headers: Hidden UX Superpowers 🦸‍♂️ ⚡

- ⏳ Rate limits (`X-RateLimit-Remaining`)
- ⏰ Deprecation & sunset (`Sunset` header)
- 🔗 Pagination links (`Link` header)
- ⚠️ Error hints and other metadata

Standards That Help (RFCs You Should Know) 📚✅

📖 RFC 7807 – Problem Details for HTTP APIs
Error format (`application/problem+json`)

🆕 RFC 6585 – Extra status codes
HTTP Status Codes (`429`, `431`, etc.)

🕰️ RFC 8594 – Sunset header for API deprecation

🌐 RFC 7231 – HTTP semantics & content
Core HTTP semantics (GET, POST, content negotiation)

Documentation: No One Likes a Mystery Box 📦❌

🔄 Keep docs updated and easy to find

🖼️ Clear field definitions

🖼️ Request and response examples

🖼️ Status codes with explanations

📄 Use OpenAPI / Swagger

📌 If you have to guess what an endpoint does, it’s broken

Versioning Without the Drama: KISS 🗂️

🔄 **Enable backward compatibility** – don’t break existing clients without a good reason.

📚 **Keep documentation in sync** – every version, every change.

🎯 **Adapt to business needs** – tech follows the strategy, not the other way around.

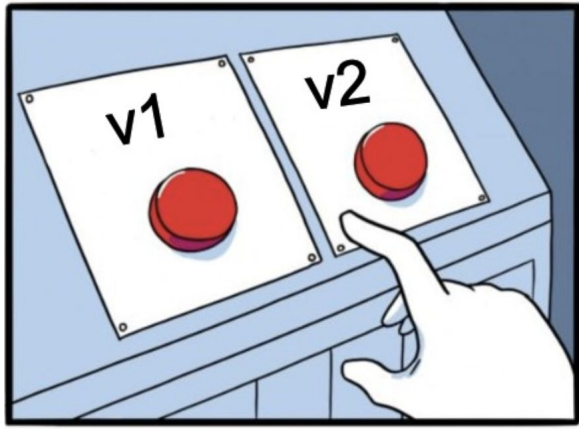
🔐 **Security first** – make sure versioning changes don’t open doors for vulnerabilities.

✅ **Non-breaking changes** — adding endpoints, adding new optional fields — don’t require bumping the major version.

Versioning Without the Drama: Golden Rule

Most APIs will start at v1 and *stay there forever*.
Only create a new major version for **breaking changes**, like:

- Changing response format
- Changing data types (e.g., integer → float)
- Removing parts of the API



Your Pragmatic API Checklist 📝✔️

🔤 Good naming, proper abstraction level

🎯 Use HTTP methods properly

📚 Paginate everything

🚨 Use structured errors

🚀 Use response headers smartly

📚 Document like a pro

🗂️ Non breaking changes, please

📌 You don’t need HATEOAS — you need **consistency**

CQRS for Pragmatic APIs

One model to rule them all? Nah.

> *“Reads and writes have different needs*
>
> *— stop forcing them to share a toothbrush.”*

Separate Write from Read Models ✍️📚

💡 Inspired by the CQRS pattern — just the useful part

📚 Read models: optimized for fetching data (no unnecessary bloat)

✍️ Write models: optimized for creating/updating (minimal and relevant fields)

🎯 Benefit: Avoid over-fetching and keep client-side logic simple

🔄 Different contexts, different shapes — it’s not duplication, it’s specialization

```
{
  "car": {
    "id": "abc-def-ghi",
    "brand": "Volvo",
    "model": "V40",
    "engine": "v1.9 -
110cv - GAS",
    "pieces": [1, 2, 3]
  }
}
{
  "offroad": {
    "id": "abc-def-ghi",
    "brand": "Volvo",
    "model": "CX90",
    "fourTractionWheels":
true,
    "extras": [1, 2, 3]
  }
}
```

```
{
  "newCar": {
    "id": "abc-def-ghi",
    "brand": "jkl-mn-opq",
    "model": "New Model to
hit the market!"
  }
}
{
  "updatedCar": {
    "brand":
"rst-abc-xyz",
    "model": "Just renamed
the model"
  }
}
```

Governance & Tooling

Designing one good API is an achievement.

Designing ten good APIs across teams?

That needs rules, tooling… and a bit of discipline.

Why API Governance 🤝🛡️

- 👥 Align teams on standards
- 🚫 Avoid chaos and duplication
- 🏃 Speed up onboarding & maintenance
- 🌟 Deliver consistent experiences

Contract-First Mindset: OpenAPI & Design Tools 🎨✍️

🧠 *Great APIs start before the first line of code.*

✅ **OpenAPI (Swagger)**

- Define request/response formats, examples, status codes
- Supports validation, mock servers, docs, and tooling

✅ **API Design Editors**

- Swagger Editor – Simple, browser-based
- Stoplight Studio – Visual OpenAPI design
- Redocly – Docs + Governance + Bundling

📌 Start with a contract, *then* implement — easier collaboration, testing, and governance.

Linting: Automate the Rules 🤖✅

✅ Use linters to catch design issues **before code review**

- **Spectral:**
  - Customizable rules for OpenAPI (naming, status codes, formats, etc.)
- **Zally:**
  - API linter with predefined corporate rulesets

🏗️ Integrate into:

- GitHub Actions / CI pipelines
- Pre-commit hooks
- Internal developer portals

📌 **Automation is your governance enforcer** — no need to manually police standards.

Final Thought ☕✨

> “APIs are promises to developers.
> Keep your promises boring, reliable, and easy to use and maintain.”

Useful Links & Resources 📚

📖 **Zalando RESTful API Guidelines** – https://opensource.zalando.com/restful-api-guidelines/

🛠️ **Zally** – Automated API linting tool – https://github.com/zalando/zally

🗂️ **OpenAPI Specification** – https://www.openapis.org/

🖊️ **Swagger Editor / OpenAPI Editor** – https://editor.swagger.io/

📘 **Redocly** – Docs that don't make your eyes bleed – https://redocly.com/

✍️ **REST Level 2 is Good Enough** – https://github.com/geeksusma/rest-2nd-level

Thanks!! 🙏

💬 Questions?