Milan Jovanović

The 5 Most Common REST API Design Mistakes (and How to Avoid Them)

9 min read · AUGUST 09, 2025

Bad APIs create friction for developers, increase maintenance costs, and make change risky. Good API design doesn't mean following every "best practice" blindly. It means choosing the right trade-offs for your context and sticking to them.

Here are 5 common mistakes I see repeatedly, why they cause problems, and how to avoid them with pragmatic, battle-tested solutions.

1. Inconsistent Naming and Structure

Naming is the first thing consumers see. Inconsistency here leads to constant documentation lookups, broken expectations, and more bugs.

Look, we've all been there. You successfully call `/users` and `/products` , so naturally you try `/orders` . But nope, this API uses `/order-list` for some reason. Now you're back to the docs, breaking your flow, wondering why anyone would do this. Multiply this friction by dozens of endpoints, and you've built an API that makes developers want to flip tables.

satisfying. They mirror your beautiful domain model! But here's the thing: you've just **hardcoded your entire data structure** into your URLs. When business requirements change (and they will), you can't reorganize without breaking clients.

Plus, what happens when someone only has a comment ID? They need to somehow figure out the user, habit, and entry IDs just to fetch one comment. That's ridiculous.

Keep it simple with **plural nouns**: `/users` , `/habits` , `/entries` . No more guessing if it's `user` or `users` .

Only nest when something truly belongs to something else. User settings belong to and die with the user, so `/users/{id}/settings` makes sense. Comments can exist independently, so `/users/{id}/posts/{postId}/comments` can be simplified.

Instead, flatten with filters:

```
# Instead of deep nesting
GET /users/{userId}/habits/{habitId}/entries

# Use filters for flexibility
GET /entries?userId={userId}&habitId={habitId}
GET /entries?habitId={habitId}  # Now you can get entries without knowing the user
```

And please, for the love of all that is holy, wrap your arrays:

```
{
  "data": [
    {
      "id": "e_8YH",
      "habitId": "code-review",
      "tags": ["team"]
    },
    {
      "id": "e_8Z2",
      "habitId": "deep-work",
      "at": "2025-08-07T07:00:00Z",
      "value": 2,
      "unit": "pomodoros",
      "note": "EF filters optimized"
    }
  ],
  "total": 42,
  "hasMore": true,
  "nextCursor": "cursor_01J9KaBcd"
}
```

I know it feels like pointless boilerplate now, but trust me, when you need to add pagination info six months from now, you'll thank yourself for not having to break every client that expects a raw array.

Yeah, the REST purists will complain that filters aren't "RESTful enough." Let them. Your API will be flexible, maintainable, and actually pleasant to use. I'll take that over conceptual purity any day.

2. Poor Versioning Strategy

Everyone defaults to **versioning** (`/v1/users`) thinking they're being smart about future changes. Spoiler: they're not. They're creating a maintenance nightmare.

Here's what actually happens when you have v1, v2, and v3 running:

- Your docs become a choose-your-own-adventure novel
- Support has no idea which version that angry customer is using
- You spend weekends maintaining code you wrote two years ago

But the worst part? **Versioning makes you lazy**. Instead of thinking "how can I evolve this without breaking clients?" you just think "eh, I'll bump the version." Now your clients have to rewrite their entire integration because you wanted to rename a field.

Watch this disaster unfold:

```
v1: GET /users returns {id, name, email}
v2: GET /users returns {id, fullName, emailAddress}  // "looks cleaner!"
v3: GET /users returns {id, firstName, lastName, email}  // "we need split names!"
```

Congrats, you're now maintaining three different response formats for the same damn data. That v1 client? They'll never get new features unless they rewrite everything. Found a critical bug? Hope you enjoy patching it three times!

Here's the **radical idea**: **don't version at all**. I'm serious.

Add fields, don't replace them:

```
// What you ship first
{ "id": 1, "name": "John Doe" }

// What you ship later (keeping the old field)
{
  "lastName": "Doe"
}
```

Need optional features? Use query parameters:

```
GET /users/{id}?include=habits,entries
GET /users/{id}?format=detailed
```

If you absolutely must make a breaking change (and really think about this), create a new resource:

```
# Old faithful, unchanged
GET /users/{id}

# New hotness
GET /userProfiles/{id}
```

When breaking changes are truly unavoidable, at least be a decent human about it. Give people 6-12 months notice. Run both versions in parallel. Write a migration guide that doesn't suck. And for crying out loud, monitor who's still using the old stuff so you can reach out before pulling the plug.

Yes, this means you need to actually think about your API design upfront. You can't just YOLO field names and fix them later. But that constraint will make you design better APIs, and future-you will buy present-you a beer.

If you want to learn more about this, I recommend reading API Change Management.

Milan Jovanović

That `GET /entries` endpoint works great with your 10 test records. Then you launch, get actual users, and suddenly you're returning 100,000 entries in a single response. Your API times out, your mobile users on crappy connections hate you, and your cloud bill makes you cry.

"We'll add pagination later," you said. Well, now it's later, and adding pagination means breaking every client that expects an array. Nice job.

Without filtering, your clients are downloading thousands of records to find the five they actually need. It's like making someone download all of Wikipedia to read one article. Your servers are melting, serializing data nobody wants. Your users are burning through their data plans. Everyone loses.

Filtering is for when you know exactly what you want:

```
GET /entries?habitId=123&date=2025-08-01&status=completed
```

Searching is for when you kinda know what you want:

```
GET /entries/search?q=morning+run+park
```

Don't try to be clever and combine them. Filtering uses your database indexes efficiently. Searching needs full-text magic. Mix them and you'll end up with something that does neither well.

For pagination, you've got two choices, and they both kinda suck in different ways.

Offset/limit is what everyone starts with:

```
          # or you can call them page and pageSize
          # use whatever you like best
```

It's dead simple and lets users jump to page 5, but here's the fun part: add or delete an item while someone's paginating, and they'll either skip entries or see duplicates. Plus, asking for offset=10000 makes your database cry as it counts through all those rows.

Cursor-based pagination is the "proper" solution:

```
GET /entries?limit=50&cursor=eyJpZCI6MTIzfQ==
```

Rock solid, no skipped items, consistent performance. But you can't jump to arbitrary pages and cursors can become invalid if underlying data changes significantly.

Yeah, implementing all this is a pain. You need cursor encoding, parameter validation, query optimization. But trying to add it after launch? Wouldn't recommend. Just build it right the first time.

4. Unclear or Inconsistent Error Handling

`{"error": "An error occurred"}` — if you return this, I hate you.

Seriously, when your API spits out these useless errors, here's what happens: I try random stuff hoping something works. I add defensive code everywhere because I don't trust you. I flood your support channel asking what's wrong. Then I complain about your API on social media (any publicity is good publicity, eh?).

Milan Jovanović

Stop inventing your own janky error format. Use **Problem Details** (RFC 9457) like a civilized developer:

```json
{
  "type": "https://api.example.com/errors/validation-failed",
  "title": "Validation Failed",
  "status": 400,
  "detail": "The request body contains invalid fields",
  "instance": "/habits/123",
  "errors": [
    {
      "field": "name",
      "reason": "Must be between 1 and 100 characters",
      "value": ""
    },
    {
      "field": "frequency",
      "reason": "Must be one of: daily, weekly, monthly",
      "value": "sometimes"
    }
  ]
}
```

See how that actually helps me fix the problem? Revolutionary, I know.

And please use the right status codes. It's not that hard:

- `400 Bad Request` : You sent garbage
- `401 Unauthorized` : Who are you?
- `403 Forbidden` : I know who you are, but no
- `404 Not Found` : That thing doesn't exist

(*I don't use this personally, and prefer returning 400 for validation failures*)

- `429 Too Many Requests` : Slow down, cowboy
- `500 Internal Server Error` : We screwed up
- `503 Service Unavailable` : We're drowning, try again later

Now, don't go leaking your entire stack trace in production like an amateur. Give friendly errors to users, detailed errors in dev/staging, and log the gory details server-side where you can actually use them.

5. Ignoring Security Until It's Too Late

"We'll add auth in phase 2" — famous last words before your API becomes a data buffet for hackers. Ask the Tea app how that worked out for them.

Here's what happens when you try to bolt on security later: Every client breaks when you add authentication. That data you've been leaking? It's probably been scraped already. Your compliance audit? Failed. That one security incident? Your users will bring it up for years.

Authentication (who are you?) and **Authorization** (what can you do?) are different things. I've seen so many APIs that check if you're logged in but never check if you should actually access that data. Don't be that person.

Rate limiting isn't just about stopping abuse, it's about fairness. Start simple: 1000 requests per hour per API key. When they hit the limit, return `429` with headers showing when they can try again. Then get fancy: different limits for different

HTTPS everywhere. Yes, even for your internal "no one will ever find this" API. It's 2025, not 2005. Let's Encrypt is free. You have no excuse.

Look, security makes things slower and more complex. Auth checks on every request, encryption overhead, state management for rate limiting, it all adds up. But you know what's worse? Explaining to your users why their data is being sold on the dark web. Build security in from the start, or prepare for a world of pain.

Final Thoughts

Good API design isn't about perfection, it's about making intentional, informed decisions. Every choice is a tradeoff. Consistency might limit flexibility. Security will impact performance. Stability means slower innovation.

Here's what actually matters: **know your tradeoffs and own them**. Document why you made these choices (future you will thank you). Stay consistent even when it's tempting not to. Design for evolution, not some imaginary perfect future. And listen to your users, but don't turn your API into a frankenstein monster trying to please everyone.

Your API is a promise to other developers. Every time you break that promise (with a breaking change, an inconsistent pattern, or a useless error message) you lose their trust. And trust me, developers hold grudges.

Build the API you'd want to use. Your developers will thank you, your support team will thank you, and honestly, you'll thank yourself six months from now when you have to maintain this thing.

Whenever you're ready, there are 4 ways I can help you:

1. **Pragmatic Clean Architecture:** Join 4,200+ students in this comprehensive course that will teach you the system I use to ship production-ready applications using Clean Architecture. Learn how to apply the best practices of modern software architecture.

2. **Modular Monolith Architecture:** Join 2,100+ engineers in this in-depth course that will transform the way you build modern systems. You will learn the best practices for applying the Modular Monolith architecture in a real-world scenario.

3. **(NEW) Pragmatic REST APIs:** Join 1,200+ students in this course that will teach you how to build production-ready REST APIs using the latest ASP.NET Core features and best practices. It includes a fully functional UI application that we'll integrate with the REST API.

4. **Patreon Community:** Join a community of 5,000+ engineers and software architects. You will also unlock access to the source code I use in my YouTube videos, early access to future videos, and exclusive discounts for my courses.

Join 70,000+ engineers who are improving their skills every Saturday morning.

Email Address

Join 70K+ Engineers