Welcome

API Design Fundamentals

Emmanuel Paraskakis

15+ years in APIs | Product Consultant for SaaS and API Companies | 3x VP PM

+ Follow

Contents

Disclaimer!

⚠️ Espouse the "Principle of Least Surprise"
⚠️ We're going for 80/20 on API Design
⚠️ The rabbit hole goes deeper
⚠️ I teach the pragmatic way
⚠️ Architects are your BFF
⚠️ Avoid bikeshedding

1. Web API Styles and when to use them

Popular Web API Styles

1. **SOAP**: Legacy, Robust
2. **REST**: Scalable, Lightweight
3. **RPC**: Internal Microservices
4. **Websockets**: Real Time, 2-Way
5. **Webhooks**: Notifications, Integrations
6. **GraphQL**: Querying, Developer-Friendly

1. SOAP (Simple Object Access Protocol)

- Older Protocol
- Robust, Secure
- Found in Enterprise (Think Banks)
- Heavyweight
- Hard to use
- Old Tooling
- XML only
- Use Cases: Legacy - Just Don't ☠️

2. REST (Representational State Transfer)

- Popular
- Scalable
- Stateless
- Lightweight
- JSON, XML, Text etc.
- Lots of tooling/frameworks
- Roy Fielding's Dissertation 
- Use Cases: Mobile, SaaS, API as a Product, AI

3. RPC (Remote Procedure Call)

- gRPC (Google) is most popular 
- Fast!
- Binary
- Efficient
- Not super usable
- Less Tooling available
- Use Cases: Internal APIs & Microservices

4. WebSockets

- Real-time
- Two-way, interactive
- Low latency, persistent connections
- Use Cases: Chat, notifications, gaming.

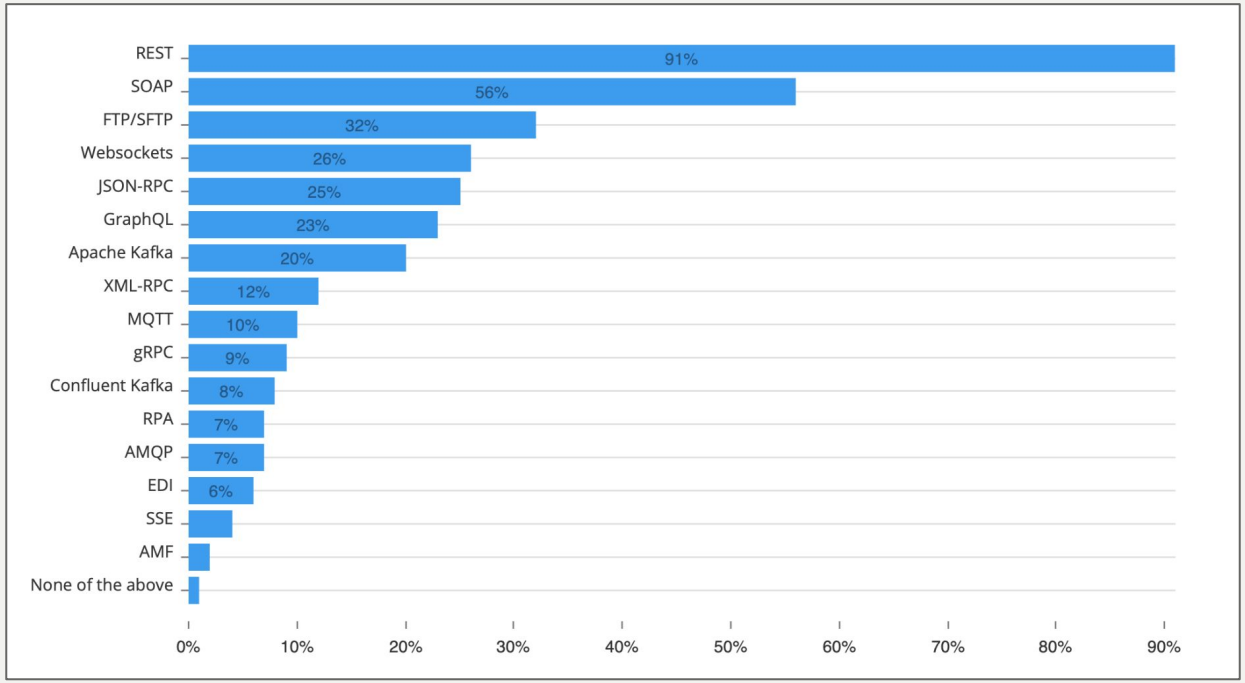
5. Webhooks

- Event-driven (sends data on trigger)
- Great for Third-party integrations
- Simple, efficient, no polling needed
- Simple, easy to understand
- May be unreliable*
- Use Cases: Notifications, Payments, Tickets

6. GraphQL

- Strongly typed!
- Efficient for front-end
- Return only the data you need
- Very usable from consumer POV
- Requires lots of set-up, complexity
- May be resource-intensive on back-end
- Use Cases: Apps with large data graphs

Which one is most popular?



SmartBear - State of Software Quality | API 2023

We'll focus exclusively on REST-ish APIs

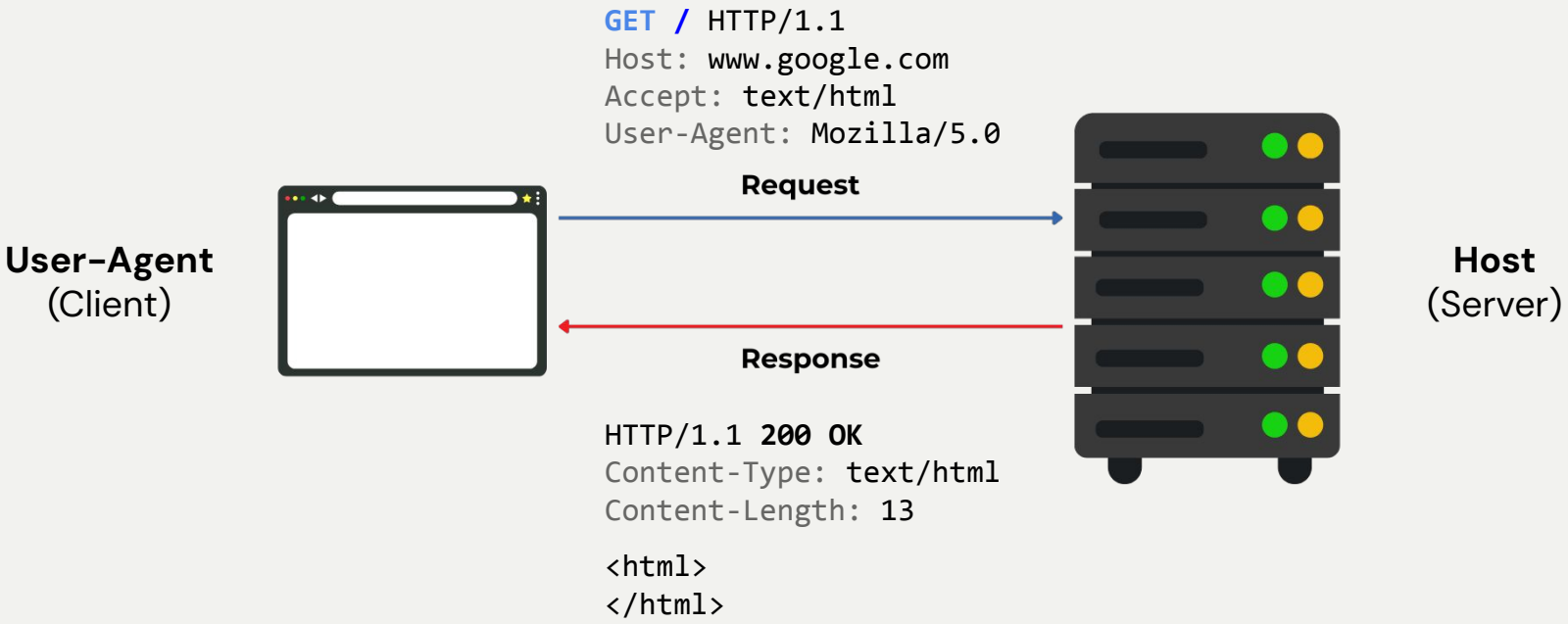
...but if you want to go deeper, here's the best analysis I've seen, from Zdeněk Němec (Z):

What API - Slides on Speakerdeck

What API - Video on YouTube

2. How APIs Work

Hypertext Transfer Protocol over a Network



Hypertext Transfer Protocol – HTTP

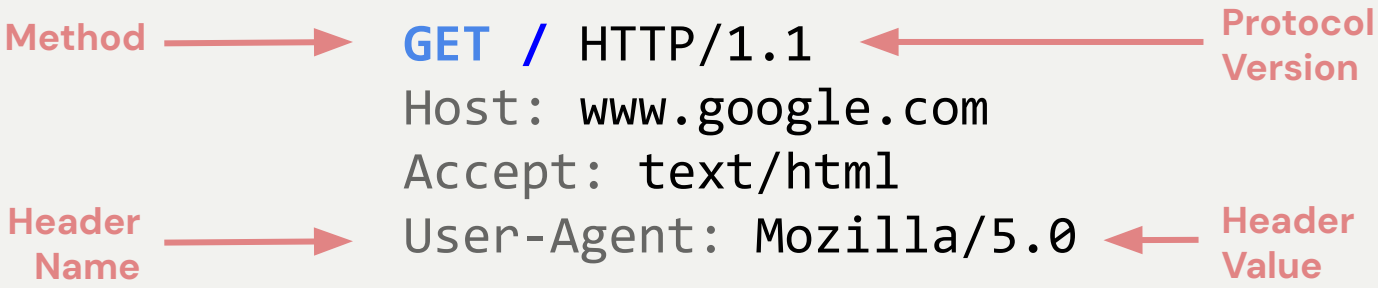
1991, Tim Berners-Lee @ CERN

Stateless application-level protocol

Foundation of the World Wide Web

- HTTP Semantics – RFC 9110
- HTTP Caching – RFC 9111
- HTTP/1.1 – RFC 9112
- HTTP/2 – RFC 9113
- HTTP/3 – RFC 9114

Request


Method
GET / HTTP/1.1
Protocol Version
Host: www.google.com
Accept: text/html
Header Name
User-Agent: Mozilla/5.0
Header Value

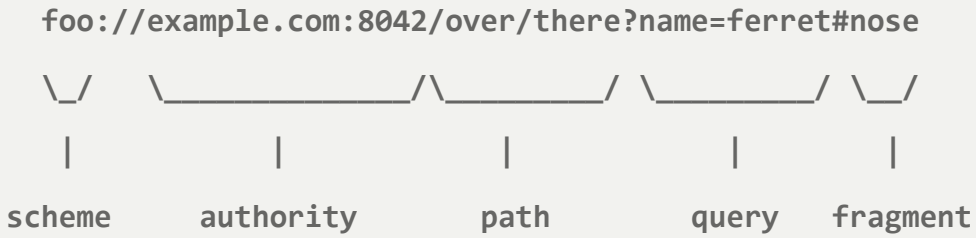
Response


Protocol Version
HTTP/1.1 200 OK
Status Code
Content-Type: text/html
Header Name
Content-Length: 13
Header Value
Response Body
<html>
</html>

Methods (Verbs)

Method	Description	Request Body*	Response Body*	Safe	Idempotent	RFC
GET	Retrieve a single resource or a collection		✅	✅	✅	RFC 9110
POST	Create a resource, or perform an action on it	✅	✅			RFC 9110
PUT	Modify an existing resource	✅	✅		✅	RFC 9110
DELETE	Remove a resource				✅	RFC 9110
PATCH	Make partial changes to an existing resource	✅	✅			RFC 5789

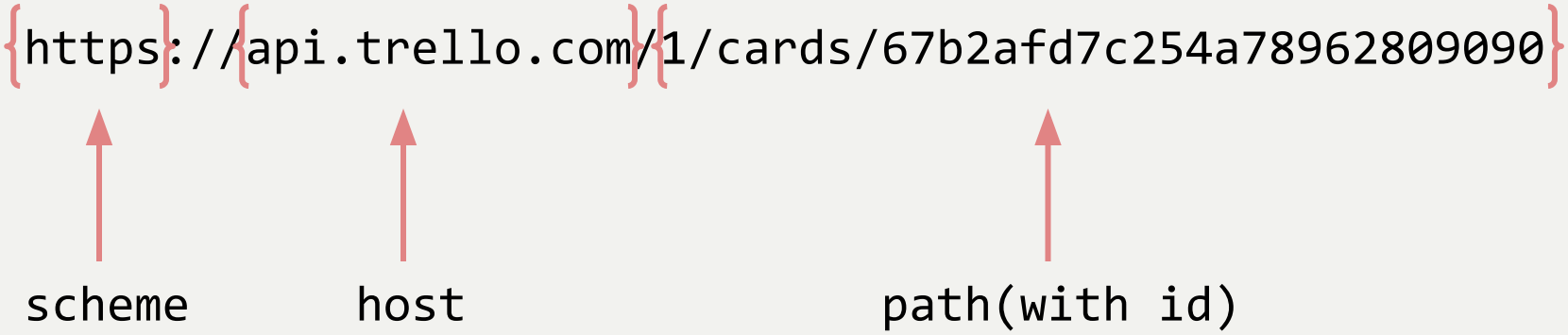
Uniform Resource Identifier – RFC 3986



Examples:

- ftp://ftp.is.co.za/rfc/rfc1808.txt
- http://www.ietf.org/rfc/rfc2396.txt
- ldap://[2001:db8::7]/c=GB?objectClass?one
- mailto:John.Doe@example.com
- news:comp.infosystems.www.servers.unix
- tel:+1-816-555-1212
- telnet://192.0.2.16:80/
- urn:oasis:names:specification:docbook:dtd:xml:4.1.2

A typical API resource – see RFC 3986



Resource Patterns (Nouns)

```
/tasks
/tasks/42
/tasks/42/status
/projects
/projects/a
/projects/a/tasks
/projects/a/tasks/42
```

```
/users
/users/emmanuel
/users/emmanuel/location
/users/emmanuel/tasks
/me
/me/location
/me/tasks
```

Verbs & Nouns Together

`GET /tasks`	Retrieve all my tasks
`DELETE /tasks/42`	Remove task 42
`GET /tasks/42/status`	Get status of task 42
`POST /projects`	Create a new project
`PUT /projects/a`	Change data in project a
`POST /projects/a/tasks`	Create a new task in project a
`GET /projects/a/tasks`	All tasks belonging to project a

Content Negotiation – RFC 9110

Client and Server can negotiate for the representation that works best for use case:

```
GET /tasks/42 HTTP/1.1
Host: www.example.com
Accept: application/json
User-Agent: Postman
```

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

{
  "id": 42,
  "title": "Buy Milk"
}
```

Content Negotiation – RFC 9110

Client and Server can negotiate for the representation that works best for use case:

```
GET /tasks/42 HTTP/1.1
Host: www.example.com
Accept: text/plain
User-Agent: Postman
```

```
HTTP/1.1 200 OK
Content-Type: text/plain

Buy Milk
```

Content Negotiation – RFC 9110

Client and Server can negotiate for the representation that works best for use case:

```
GET /tasks/42 HTTP/1.1
Host: www.example.com
Accept: application/pdf
User-Agent: Postman
```

```
HTTP/1.1 406 Not Acceptable
Content-Type: application/problem+json

{
    "title": "Not Acceptable",
    "status": 406,
     "detail": "Only 'text/plain or
'application/json' content types supported"
}
```

Status Codes –Success– RFC 9110

200 OK	General success status
201 Created	Resource created successfully
202 Accepted	Accepted for later processing
204 No Content	Success but no data returned
206 Partial Content	In Response to Range requests

Status Codes –Client Error– RFC 9110, RFC 6585

400 Bad Request	Bad or unexpected data in request
401 Unauthorized	Authentication needed
403 Forbidden	Insufficient permissions
404 Not Found	Resource is unavailable
405 Method Not Allowed	Use a different method
406 Not Acceptable	Ask for other media type
415 Unsupported Media Type	Does not understand media type
429 Too Many Requests	Rate limited, may try again

Status Codes –Server Error– RFC 9110

500 Internal Server Error	General server problem
502 Bad Gateway	General network problem
503 Service Unavailable	Server may overloaded or down

How to think about HTTP

- **Resource** → Identity (Noun)
- **Method** → Action (Verb)
- **Status** → Result
- **Body** → Request or Response Content
- **Headers** → Metadata
- **Query Parameters** → Modify Response

3. Naming, Vocabulary, and Data Types / Structures

Pragmatic Naming

- ❑ Have a glossary
- ❑ Check schema.org
- ❑ Be consistent in naming
- ❑ Be consistent in casing*
  (`camelCase`, `kebab-case`, `snake_case`)
- ❑ Do not abbreviate, don't use acronyms…
  …except very well known terms, for example "NASA"
- ❑ Pluralize nouns in resources: `/tasks`, not `/task`
- ❑ Use normal language, it must flow (US English 😇)

Bad → Good Examples

`lob`	`lineOfBusiness`
`dateLastActivity`	`updatedAt`
`idList`	`listID`
`isComplete`	`completed`
`unitCode`	`unit`
`zip`	`postalCode`

Resources

- ❑ Nouns*
- ❑ Case-Insensitive
- ❑ Pluralize collections
- ❑ Don’t worry about URI length (anymore)*
- ❑ If you need to separate words use dash “-”
- ❑ Don’t nest more than 2-5 levels (inc. params)
  I.e. no more than `/projects/a/tasks/42/status`

Query parameters

- e.g. `/tasks?status=active&year=2025`
- Urlencode (space becomes `%20`)
- Case-Insensitive
- Use filtering, sorting, pagination etc...
- But don't go crazy - consider GraphQL instead
- If too many params consider a **POST** body

Headers

- Hyphenated-Pascal-Case
- Custom Headers: start with “X-”

Representing Data in REST APIs

Format	Media Type (IANA)	Use Case
JSON	`application/json`	Most APIs will use JSON
XML	`application/xml`	Older APIs may use XML
HTML	`text/html`	Web pages!
Plain Text	`text/plain`	Useful for short messages
Short	`application/x-www-form-urlencoded`	Web forms
Form Data	`multipart/form-data`	Longer forms & Binary data
PNG	`image/png`	Pictures & Photos
Problem Details	`application/problem+json`	Standard way to output errors

In this presentation we'll focus on JSON

- Structured
- Readable/Efficient (it's just text)
- Easy to Use in Code (JavaScript)

but...

- Writing! Mind the brackets, commas, quotes ("≠")
  - **Use JSON Lint! Don't skip!** ⚠️
  - Or a Code Editor
- Not Strongly Typed

Key Names

- Be consistent in casing:
  (`camelCase`, `kebab-case`, `snake_case`)
- Only use:
  - ASCII alphanumeric characters
  - underscore (_)
  - dollar sign ($)

Data as key/value pairs

```
{
  "firstName": "Emmanuel",
  "middleName": null,
  "lastName": "Paraskakis",
  "instructor": true,
  "cohortCount": 1
}
```

Objects

```
{
  "name": {
    "first": "Emmanuel",
    "middle": null,
    "last": "Paraskakis"
  },
  "instructor": true,
  "cohortCount": 1
}
```

Arrays

```
{
  "courses": [
    {
    "name": "API Fundamentals for Product Managers",
    "cohort": "Cohort 1"
    },
    {
    "name": "API Mastery for Experienced PMs",
    "cohort": "Cohort 1"
    }
  ]
}
```

Nesting

- ❏ Nest but just enough 😭
- ❏ Flatter is better for LLMs
- ❏ Consider using links to resources rather than including
  - ❏ Over-fetching: payloads larger than needed
  - ❏ Under-fetching: user makes too many calls
  - ❏ Consider GraphQL

4. Conventions and Best Practices

Retrieve a Collection

```
GET /tasks?year=2025
```

Request

- ❑ Empty body
- ❑ Query parameters (if too many, make it a post)

Response

- ❑ All members of collection that match as array
- ❑ **200** for success, **404** for not found

Request

N/A

Response

```
[
  {
  "id": 42,
  "name": "Buy Milk",
  "completed": false
  },
  {
  "id": 47,
  "name": "Buy Cookies",
  "completed": true
  }
]
```

Retrieve a Single Resource

`GET /tasks/42`

Request

- ❑ Empty body
- ❑ No query parameters

Response

- ❑ Single resource representation
- ❑ **`200`** for success, **`404`** for not found

Request

N/A

Response

```
{
  "id": 42,
  "name": "Buy Milk",
  "completed": false
}
```

Create a Resource

POST `/tasks`

Request

- ❑ JSON Body
- ❑ No query parameters

Response

- ❑ Single resource representation, if successful
- ❑ Location header with full resource
- ❑ **201** for success, **4XX** depending on reason

Request

```
{
  "name": "Buy Milk",
  "completed": false
}
```

Response

```
{
  "id": 42,
  "name": "Buy Milk",
  "completed": false
}
```

Remove a Single Resource

`DELETE /tasks/42`

Request

- ❑ Empty Body
- ❑ No query parameters

Response

- ❑ Empty Body
- ❑ **204** for success, **404** if not found

Request

N/A

Response

N/A

Change a Single Resource

PUT `/tasks/42`

Request

- ❑ JSON Body
- ❑ No query parameters

Response

- ❑ Resource Representation
- ❑ **`200`** for success, **`4XX`** depending on what went wrong

Request

```
{
  "name": "Buy Milk",
  "completed": true
}
```

Response

```
{
  "id": 42,
  "name": "Buy Milk",
  "completed": true
}
```

DO NOT:

❌ Invent your own conventions – “Least Surprise”
❌ Abuse `POST` with RPC – `POST /deletetask`
❌ Use `PATCH`, unless you are Google 😜
❌ Overthink it, unless you’re into it 🗓️
❌ Create unneeded complexity!
❌ Introduce breaking changes
❌ Return **`200`** for everything
❌ Be inconsistent

5. API Design Patterns

How to Design a Good API and Why it Matters

How to Design a Good API and Why it Matters

Joshua Bloch
Principal Software Engineer
Google™

1 How to Design a Good API and Why it Matters

Characteristics of a Good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

Use UUIDs for Identifiers – RFC 4122

✅ Must be Unique!

✅ Use UUID like:

`16556f6a-64cc-4522-89aa-c9f843c100a8`

✅ Server generated

❌ Avoid predictable, consecutive IDs like 42

❌ Do not leak user/private info that can be abused by an attacker

Use only ISO-8601 for Date/Time - RFC 3339

Use ISO-8601:

- E.g. **`2025-02-20T19:17:53Z`**
- Store and return “Z” or GMT/UTC

Less common, Unix epoch:

- E.g. `1740005740`
- Seconds since 1 January 1970 😏

Filtering

```
GET /tasks?year=2025&completed=true
```

Should return a collection of only the items that match

⚠️ Smell: Don’t go crazy with this, use GraphQL instead

(especially if you need to start using operators or to change the shape of the response to exclude some fields or include referenced resources)

Sorting

```
GET /tasks?sort=year&order=desc
```

Should return a collection in the requested order

⚠️ Order is normally undefined in collections

Pagination (easy mode)

```
GET /tasks?offset=20&limit=10
```

Should only return 10 items starting with the 20th

You can include links to previous/next and first/last pages in the response.

Use “Problem Details” for Errors – RFC 9457

Use the application/problem+json media type

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json
{
  "type": "https://example.net/validation-error",
  "title": "Your request parameters are not valid.",
  "invalid-parameters": [
    {
      "name": "age",
      "reason": "must be a positive integer"
    }
  ]
}
```

🚨 Versioning – The Problem 🚨

1. Customers have integrated your API
2. You want to evolve the API
3. They don’t want to rewrite code
4. You don’t want customers to churn

🚨 Versioning – The (Ugly) Solution 🚨

1. Keep the old API around
2. Release a new API
3. Use *something* to identify the new version
4. You now have 2 APIs to support, maintain

🚨 Versioning – The (Ugly) *something* 🚨

1. `/v2` in URL
2. Custom header like `X-API-Version: 2`
3. Custom media type:
   `Accept: application/vnd.mytype.v2+json`

❌ Do not use query parameters like `?version=1`
❌ Do not use full semver, just major version
❌ Do have a deprecation/sunsetting policy
❌ Do not introduce breaking changes*

Non-Breaking Changes:

✅ Adding a new resource: `/milestones`
✅ Adding an optional field: `"priority"`
✅ Adding an optional parameter: `?priority=`
✅ Changing from required to optional

Example: Need to split `"name"` field into `"firstName"` and `"lastName"` – retain the old field

6. Authentication Types and Standards

Concepts

- ❑ Authentication: Who am I?
- ❑ Authorization: What do I have access to?
- ❑ Least Privilege: Give me only what I need
- ❑ Encryption: Others can’t eavesdrop
- ❑ Availability: Fair use and DoS prevention

Top 7 API Authentication Methods Compared

Method	Best Use Case	Key Strength	Limitations
OAuth 2.0	Third-party integrations	Fine-grained access control	Complex setup, resource-heavy
API Keys	Internal services or Public APIs	Easy to implement	Limited security, no expiration
JWT	Microservices	Stateless, fast performance	No revocation, large token size
~~**Basic Authentication**~~	~~Legacy systems~~	~~Simple setup~~	~~High security risk, relies on HTTPS~~
Bearer Authentication	Modern web APIs	Token-based, scalable	Requires token management
mTLS	High-security systems	Mutual authentication with certificates	Complex certificate management
OpenID Connect	Identity management	Combines authentication and authorization	Steep learning curve, detailed configuration

How to Choose

- ➔ **High Security?** Use OAuth 2.0 or mTLS.
- ➔ **Scalability?** JWT or Bearer Authentication.
- ➔ **Simplicity?** API Keys or ~~Basic Authentication~~.
- ➔ **Identity Management?** OpenID Connect.
- ➔ **Developer Experience?** API Keys.

Zuplo: Top 7 API Authentication Methods Compared

API Key

- ❑ Generated
- ❑ Popular and easy
- ❑ Suitable for automation
- ❑ Pass in with every request

but...

🚨 Can be misplaced, like a physical key

DO NOT:

❌ Pass sensitive info like a key or token in the URL ☠️
❌ Prefer custom header vs `Authorization` ☠️
❌ Use Basic Authentication or OAuth 1.0 ☠️
❌ Use plain `http` - use `https` instead ☠️
❌ Disregard Monitoring & Alerting ☠️
❌ Roll your own Auth/Encryption ☠️
❌ Ignore the OWASP API Top-10 ☠️
❌ Store secrets insecurely ☠️
❌ Skip an API Gateway ☠️
❌ Forget Rate Limiting ☠️