Designing Web APIs

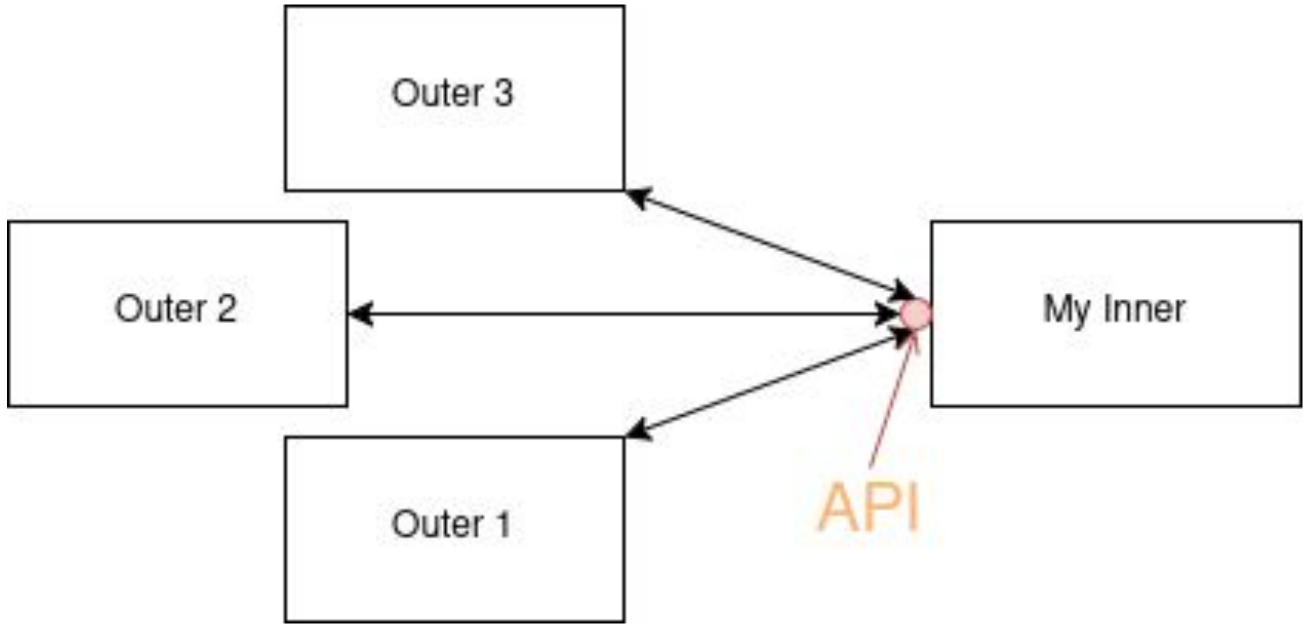
Pham Thanh Vinh - Truong Thanh Toan

2025 - fit@hcmus

What are APIs?

Application **P**rogramming **I**nterface

What are APIs?



How components interact

RESTful API - REST principles

- Uniform Interface
- Client - Server
- Stateless
- Cacheable
- Layered System

RESTful API - REST principles

- Uniform Interface
- Client - Server
- Stateless
- Cacheable
- Layered System

Problems

- How to design an *easy-to-use* API?
- What if the API *grows*?
- *Security* issues you may encounter!

Content

- Best practice
- Documentation
- Versioning
- Security

Content

- Best practice
- Documentation
- Versioning
- Security

Best Practice: Resource oriented design

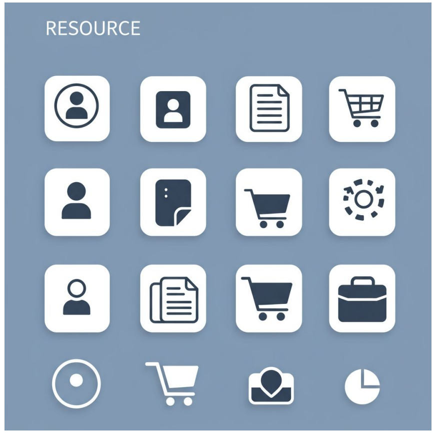
What is Resource oriented design

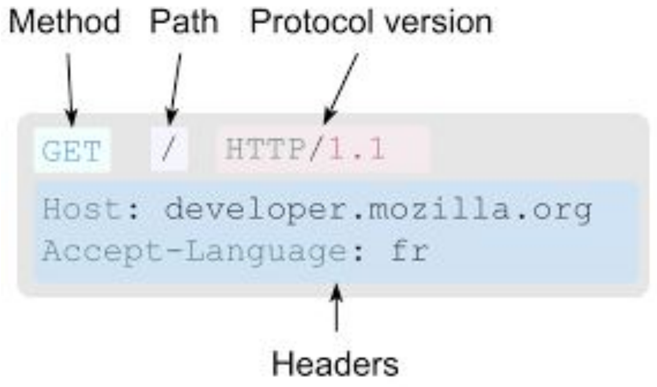
Shifts focus from *actions* (verbs) to *things* (nouns/resources).

Models the system as a collection of manageable resources.

Each resource has a unique identifier.

Uses a standard interface (HTTP) to interact with resources.





```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Best Practice: Resource oriented design

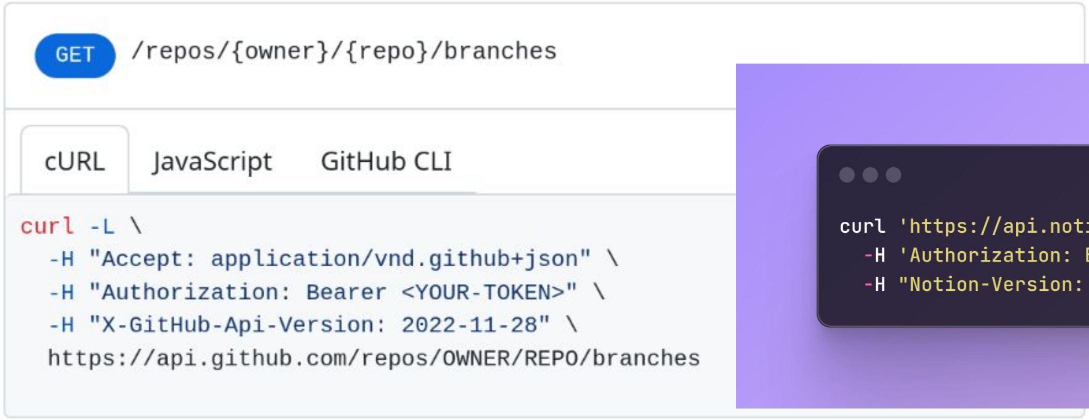
Examples: URI

Use Nouns, Not Verbs: `/users`, NOT `/getUsers` (Action is from HTTP method).

Use Plural Nouns for Collections: `/users`, `/orders`, `/products`.

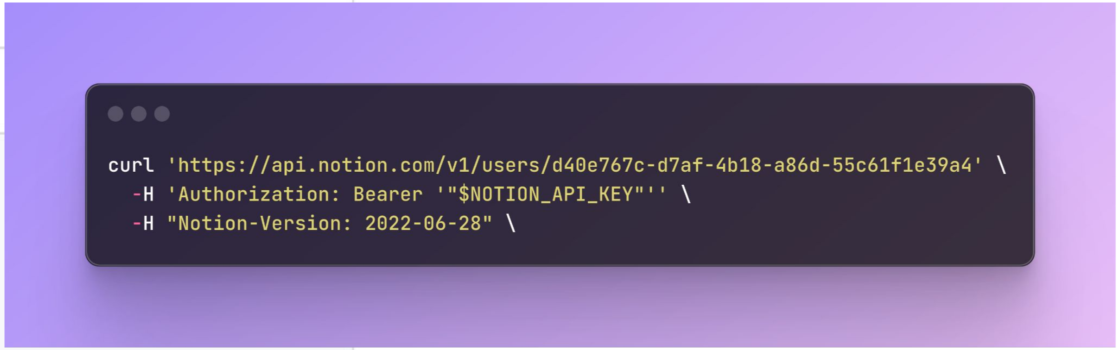
Access Items via ID: `/users/123`, `/orders/45`

Avoid Trailing Slashes: `/users`, not `/users/`



```
/repos/{owner}/{repo}/branches
```

```
curl -L \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/OWNER/REPO/branches
```



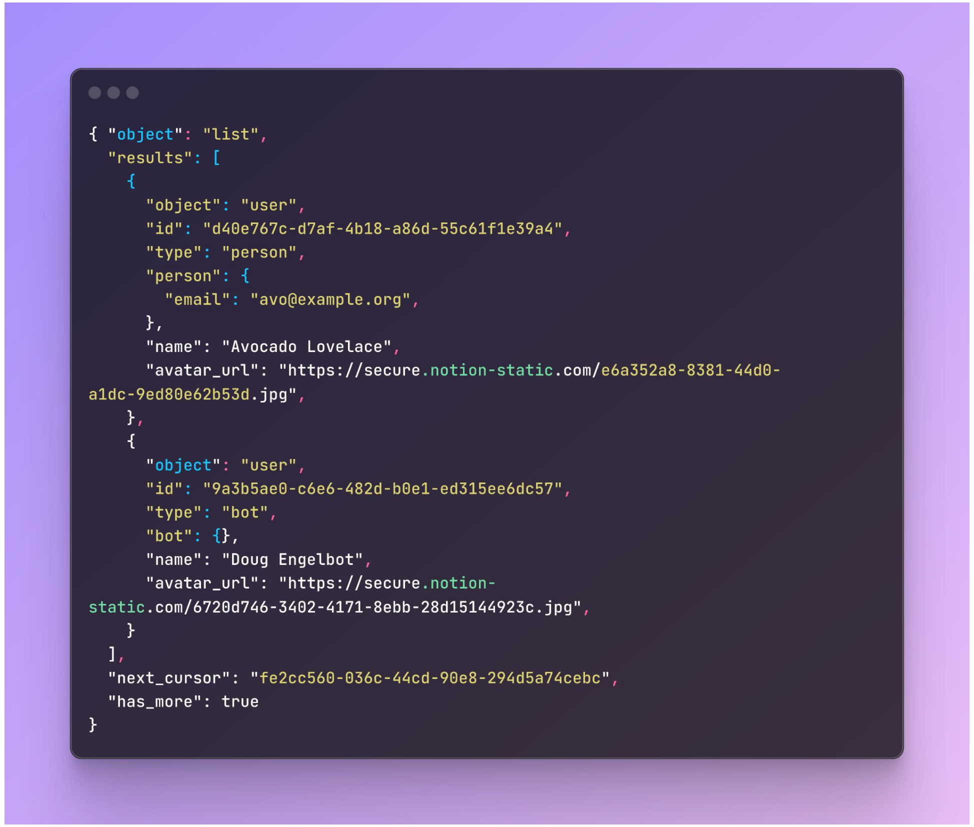
```
curl 'https://api.notion.com/v1/users/d40e767c-d7af-4b18-a86d-55c61f1e39a4' \
  -H 'Authorization: Bearer '"$NOTION_API_KEY"'' \
  -H "Notion-Version: 2022-06-28" \
```

Best Practice: Resource oriented design

Examples: Resource Representation

A representation captures the resource's state *at a specific time*.
Includes:

- Data (the actual info)
- Metadata (data about the data)
- Hypermedia Links (HATEOAS - links to related resources/actions)



```
{ "object": "list",
  "results": [
    {
      "object": "user",
      "id": "d40e767c-d7af-4b18-a86d-55c61f1e39a4",
      "type": "person",
      "person": {
        "email": "avo@example.org",
      },
      "name": "Avocado Lovelace",
      "avatar_url": "https://secure.notion-static.com/e6a352a8-8381-44d0-
a1dc-9ed80e62b53d.jpg",
    },
    {
      "object": "user",
      "id": "9a3b5ae0-c6e6-482d-b0e1-ed315ee6dc57",
      "type": "bot",
      "bot": {},
      "name": "Doug Engelbot",
      "avatar_url": "https://secure.notion-
static.com/6720d746-3402-4171-8ebb-28d15144923c.jpg",
    }
  ],
  "next_cursor": "fe2cc560-036c-44cd-90e8-294d5a74cebc",
  "has_more": true
}
```

Best Practice: Resource-Oriented Design

Why Resource-Oriented Design Matters

Simplicity: Easy to understand and use.

Predictability: Developers can often guess endpoints.

Leverages HTTP: Builds on existing web standards.

Scalability & Maintainability: Clear structure helps growth.

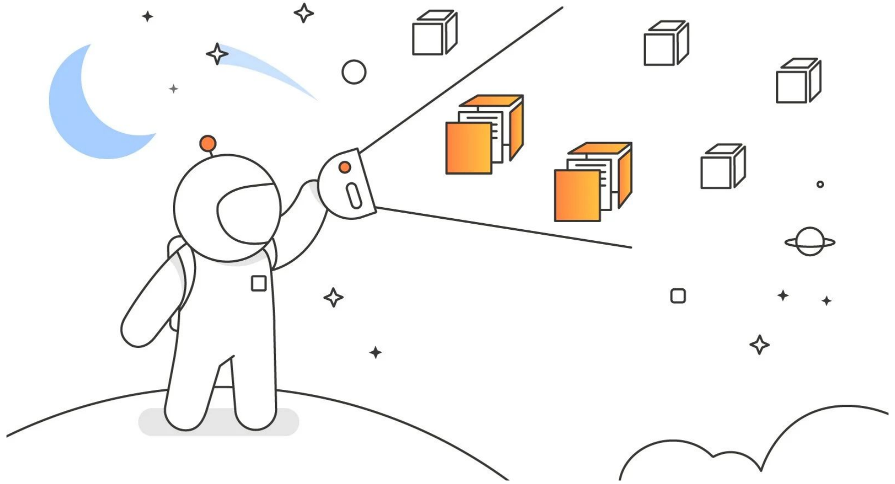
Best Practice: Handling Collections

Working with Lists of Resources

Problem: Retrieving *all* items from a large collection (e.g., `/users`, `/products`) is often slow and inefficient.

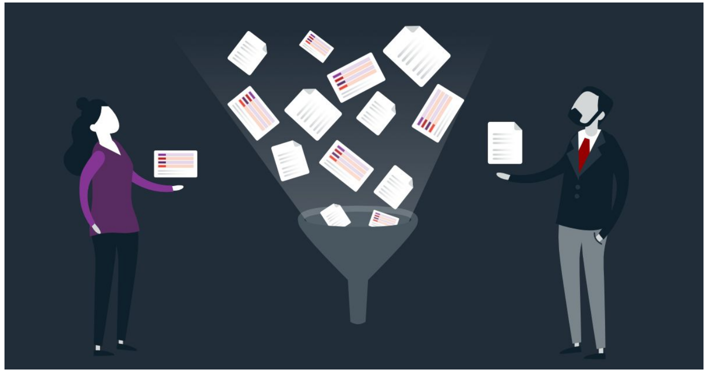
Solution: Need ways to request specific subsets:

- **Filtering:** Get items matching criteria.
- **Sorting:** Order the items.
- **Pagination:** Get items in chunks (pages).



Best Practice: Handling Collections

Filtering: Getting Only What You Need



What: Request only resources matching specific criteria.
How: Query parameters mapping field names to values.

- *Example:* `GET /items?state=active&seller_id=1234`

Advanced:

- Operators (greater than, less than): `?price[gte]=10` or `?price=gte:10`
- Multiple values: `?status=pending&status=shipped` or `?sizes=M,L`
- Logical AND/OR (often implicit or via specific `filter` parameter).

Best Practice: Use clear names, document options, be consistent!

Best Practice: Handling Collections

Sorting: Ordering the Results

What: Specify the order of resources returned.
How: `sort` or `order_by` query parameter.

- *Example:* `?sort=price`

Direction: Indicate ascending/descending.

- *Prefix Example:* `?sort=-price` (descending), `?sort=+name` (ascending)
- *Suffix Example:* `?sort=id:desc`

Multiple Fields: Comma-separated list (applied sequentially).

- *Example:* `?sort=lastName,+firstName`

Best Practice: Provide a default sort order. Document sortable fields.

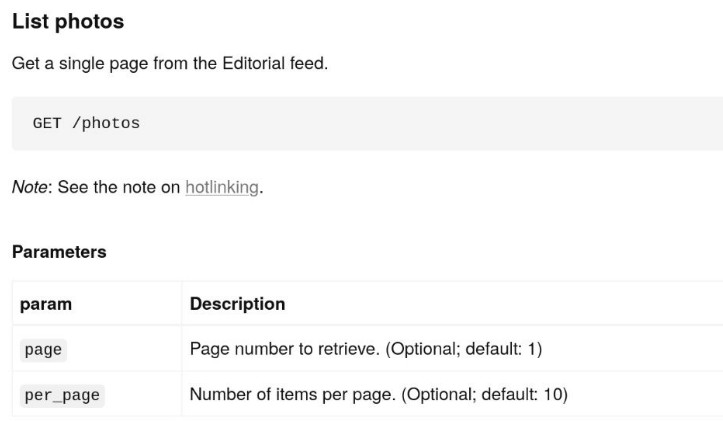
Best Practice: Handling Collections

Pagination - Don't Fetch Everything at Once!

What: Break down large result sets into smaller "pages".

Why: Improves performance, reduces load, better user experience.

CRITICAL: Design pagination in from the start – adding it later often breaks compatibility!



List photos

Get a single page from the Editorial feed.

```
GET /photos
```

Note: See the note on hotlinking.

Parameters

param	Description
`page`	Page number to retrieve. (Optional; default: 1)
`per_page`	Number of items per page. (Optional; default: 10)

Use standard parameter names (`limit`, `offset`, `page`, `size`, `after`, `before`, `cursor`).

Include pagination metadata in the response (e.g., total items, links to `next`/`prev` pages).

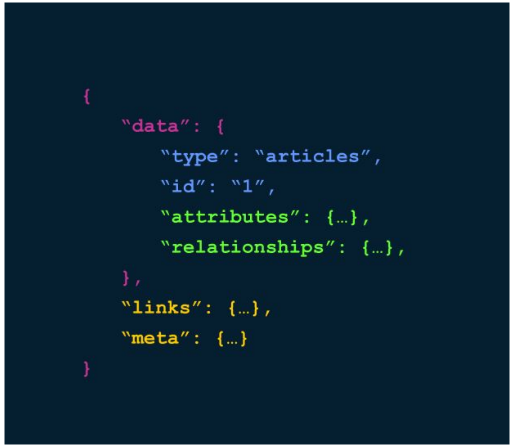
Often uses the `Link` HTTP header.

Allow clients to request page size (but set reasonable server-side limits).

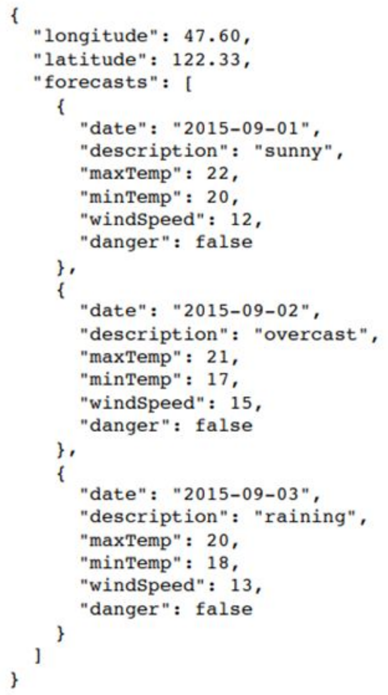
Best Practice: JSON

JSON: The Language of Modern APIs

Consistency is King: Apply consistent structure and naming conventions.
Use Top-Level Objects: Allows for metadata.
Handle Types Deliberately: Be clear about `null` vs. omitted, use ISO 8601 for dates, strings for large numbers/enums.
Document Your Choices: Help developers understand your API's JSON format.



```
{
    "data": {
        "type": "articles",
        "id": "1",
        "attributes": {…},
        "relationships": {…},
    },
    "links": {…},
    "meta": {…}
}
```



```
{
  "longitude": 47.60,
  "latitude": 122.33,
  "forecasts": [
    {
      "date": "2015-09-01",
      "description": "sunny",
      "maxTemp": 22,
      "minTemp": 20,
      "windSpeed": 12,
      "danger": false
    },
    {
      "date": "2015-09-02",
      "description": "overcast",
      "maxTemp": 21,
      "minTemp": 17,
      "windSpeed": 15,
      "danger": false
    },
    {
      "date": "2015-09-03",
      "description": "raining",
      "maxTemp": 20,
      "minTemp": 18,
      "windSpeed": 13,
      "danger": false
    }
  ]
}
```

Best Practice: Ensuring Reliability - Idempotency

Idempotency - Making Requests Safe to Retry

What is it? An operation where making the *same request* multiple times has the *same effect* on the server's state as making it just once.

Best Practice: Ensuring Reliability - Idempotency

Which Methods Are Naturally Idempotent?

HTTP Defines Some Methods as Idempotent:

- ✅ **Idempotent:** `GET`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`, `TRACE`
- ❌ **Not Naturally Idempotent:** `POST`, `PATCH`

Solution: Verify post and patch data before making change

Best Practice: Optimizing Performance: API Caching

Caching - Making Your API Faster

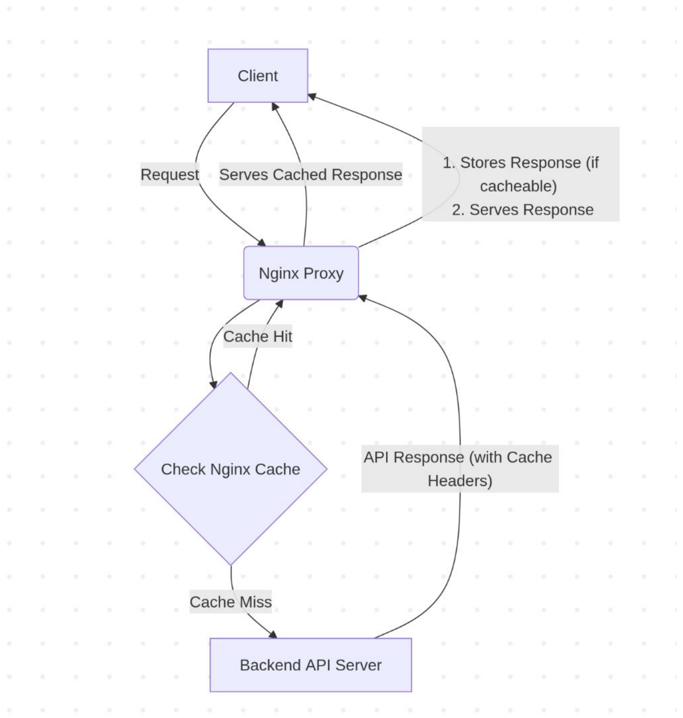
Faster Responses: Lower latency for users.

Reduced Server Load: Less work for your backend systems.

Bandwidth Savings: Avoid re-sending the same data.

Best Practice: Optimizing Performance: API Caching

Proxy Caching: Nginx Example



See u in the demo

Best Practice: Optimizing Performance: API Caching

Proxy Caching: Nginx Example

https://github.com/vicyan1611/api_seminar

Content

- Best practice
- Documentation
- Versioning
- Security

Why API documentation?

What did ***they*** do?

Why API documentation?

What did ***I*** do?

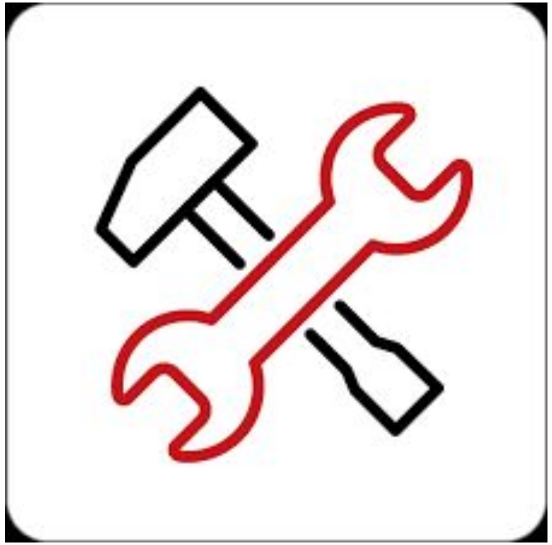
Why API documentation?

- Developer Experience (DX)

What did ***I*** do?

Why API documentation?

- Developer Experience (DX)
- Maintainability



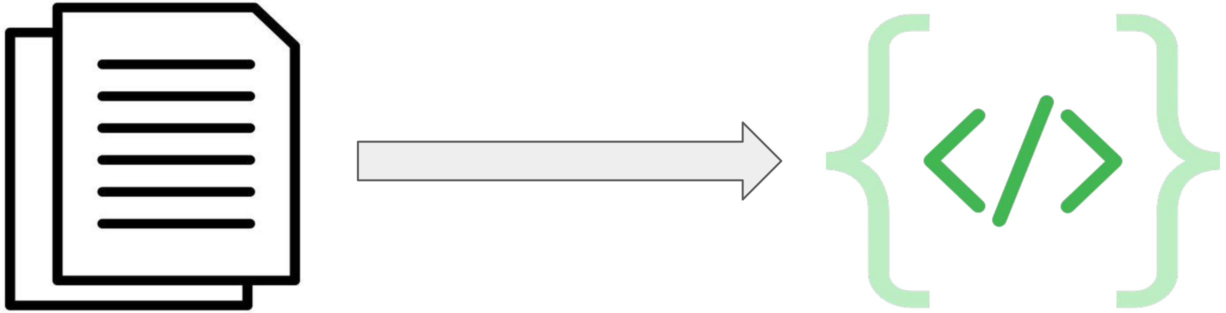
Why API documentation?

- Developer Experience (DX)
- Maintainability
- Awareness

Microsoft

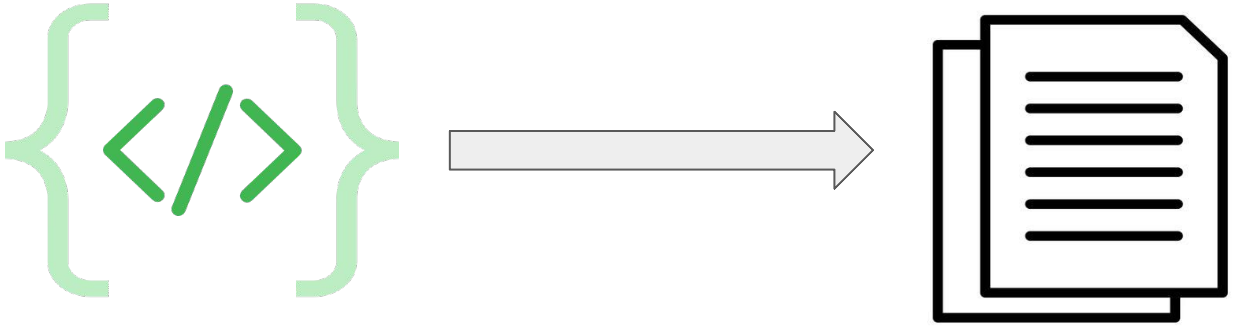
Approaches

- Document First



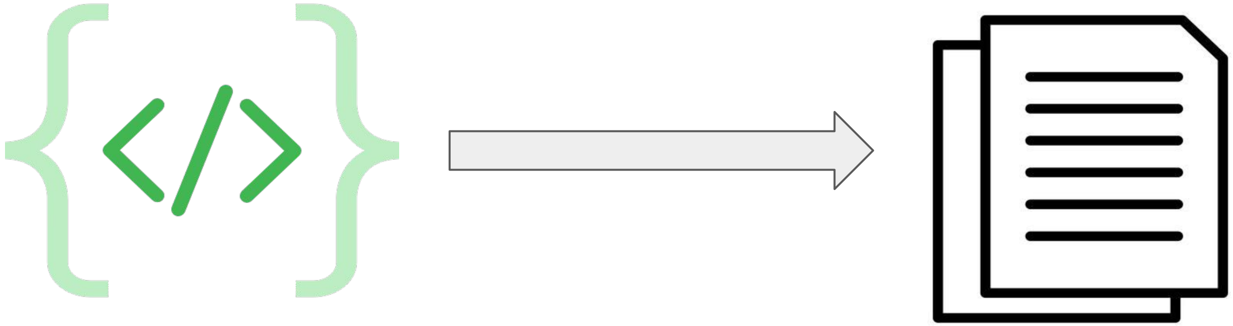
Approaches

- Document First
- Code First



Approaches

- Document First
- Code First



OpenAPI

- OpenAPI Specifications
- OpenAPI Definition

OpenAPI – the document aspect

- OpenAPI Specifications
- OpenAPI Definition
- Template: click here

What can we do with the documents

What can we do with the documents

- Automation testing

What can we do with the documents

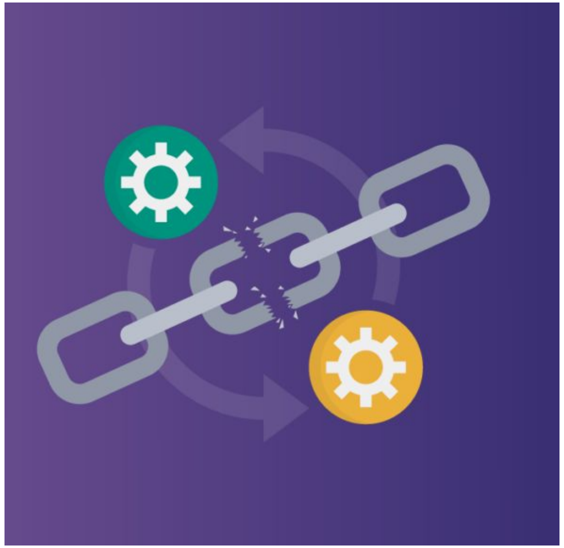
- Automation testing
- Generate SDK

What can we do with the documents

- Automation testing
- Generate SDK
- Create Server Boilerplate

Content

API evolution



Breaking changes

New features

Bug fixes

Options

Alter the old source code

Options

$$\begin{tikzcd}
\text{Alter the old source code} \arrow[d] \\
\text{Incompatibility}
\end{tikzcd}$$

Options

~~Alter the old source code~~

↓

~~Incompatibility~~

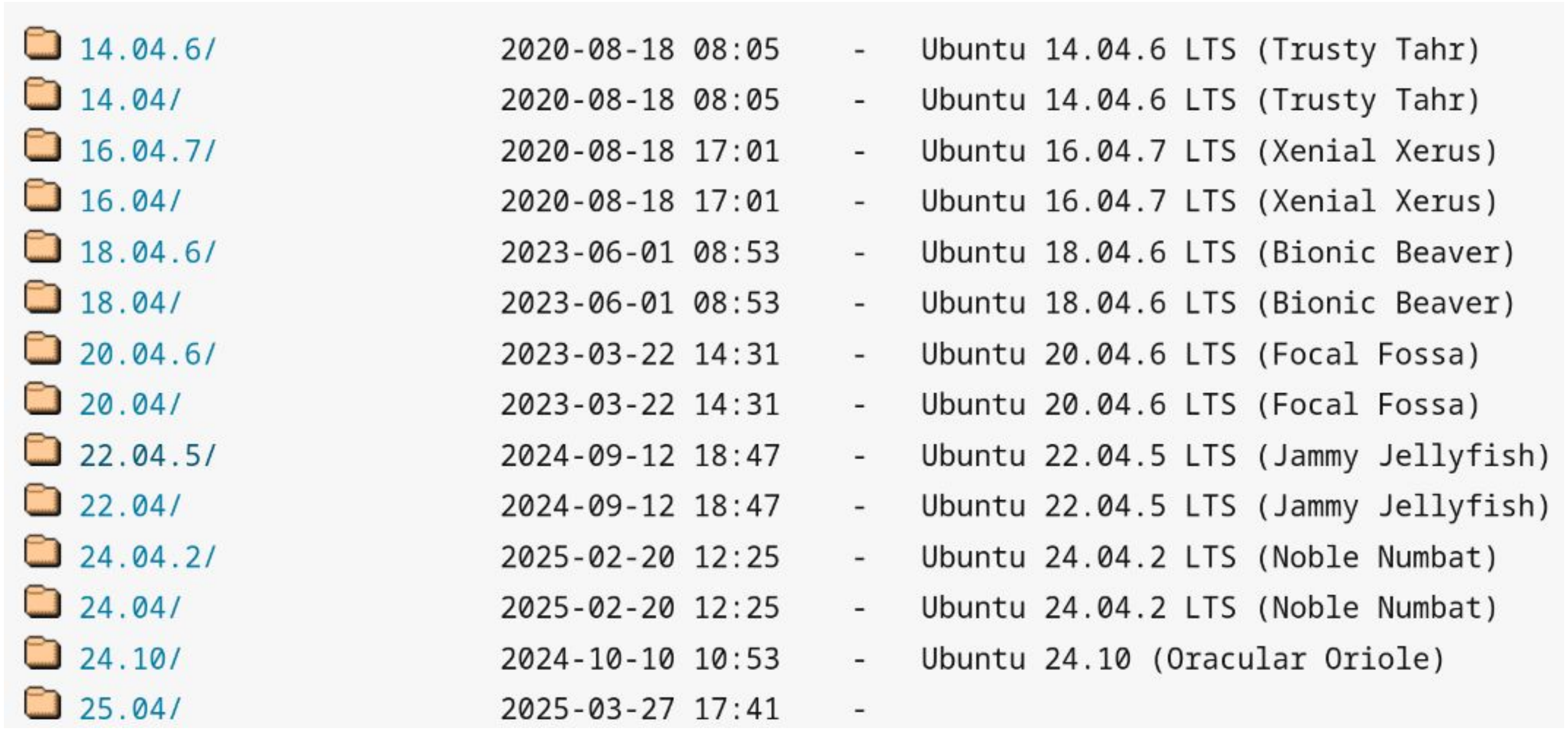
→ **Versioning**

Numbering

Numbering: Date-based

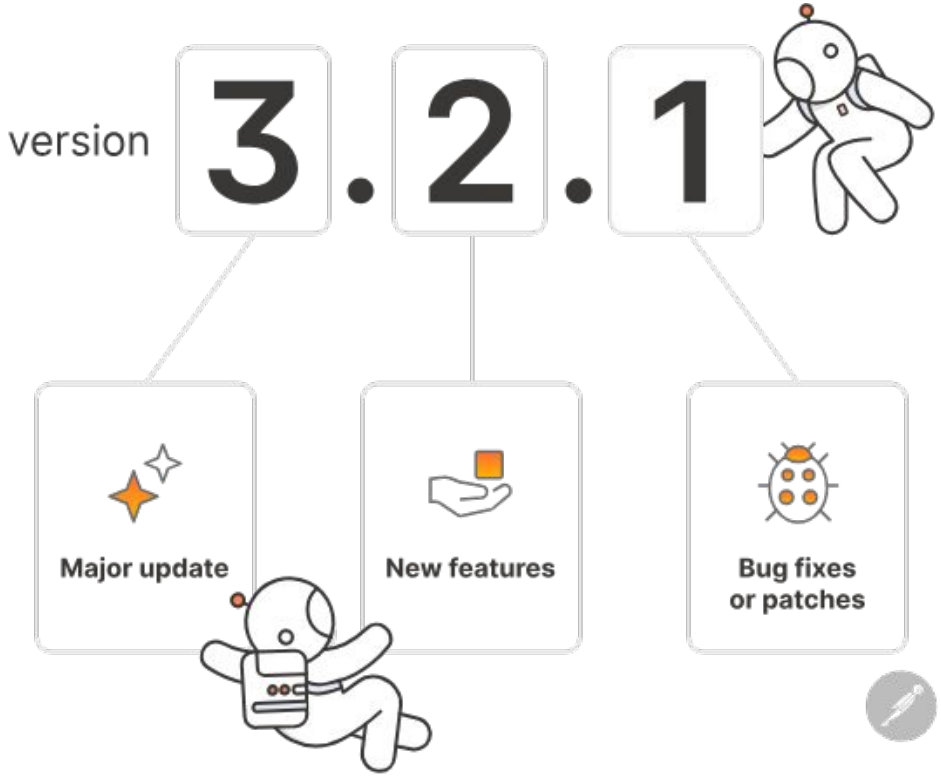
$$\underbrace{\texttt{2020}}_{\texttt{year}}\texttt{.}\underbrace{\texttt{1}}_{\texttt{release}}\texttt{.}\underbrace{\texttt{4018}}_{\texttt{build}}\underbrace{\texttt{-pre}}_{\texttt{status}}$$

Numbering: Date-based



```
14.04.6/    2020-08-18 08:05  -  Ubuntu 14.04.6 LTS (Trusty Tahr)
14.04/      2020-08-18 08:05  -  Ubuntu 14.04.6 LTS (Trusty Tahr)
16.04.7/    2020-08-18 17:01  -  Ubuntu 16.04.7 LTS (Xenial Xerus)
16.04/      2020-08-18 17:01  -  Ubuntu 16.04.7 LTS (Xenial Xerus)
18.04.6/    2023-06-01 08:53  -  Ubuntu 18.04.6 LTS (Bionic Beaver)
18.04/      2023-06-01 08:53  -  Ubuntu 18.04.6 LTS (Bionic Beaver)
20.04.6/    2023-03-22 14:31  -  Ubuntu 20.04.6 LTS (Focal Fossa)
20.04/      2023-03-22 14:31  -  Ubuntu 20.04.6 LTS (Focal Fossa)
22.04.5/    2024-09-12 18:47  -  Ubuntu 22.04.5 LTS (Jammy Jellyfish)
22.04/      2024-09-12 18:47  -  Ubuntu 22.04.5 LTS (Jammy Jellyfish)
24.04.2/    2025-02-20 12:25  -  Ubuntu 24.04.2 LTS (Noble Numbat)
24.04/      2025-02-20 12:25  -  Ubuntu 24.04.2 LTS (Noble Numbat)
24.10/      2024-10-10 10:53  -  Ubuntu 24.10 (Oracular Oriole)
25.04/      2025-03-27 17:41  -
```

Numbering: Semantic-based



Methods

Methods: URL Path

Legacy (v1.1)

`https://api.twitter.com/1.1/statuses/user_timeline.json`

Latter (v2)

`https://api.twitter.com/2/users/me`

Methods: URL Path

✅	❌
• Visible • Easy to test on browser • Caching	• Dirty URLs • Static version • Require big changes

Methods: Query Parameters

Legacy (v1.0)

```
https://api.example.com/resources?version=1.0
```

Latter (v2.0)

```
https://api.example.com/resources?version=2.0
```

Methods: Query Parameters

✅	❌
• Clean base URL • Default to latest	

Empty version param:

`https://api.example.com/resources`

Methods: Query Parameters

✅	❌
● Clean base URL ● Default to latest	● URL clutter, less discoverable

```
https://api.example.com/resources?&name=hehe&age=omg&tok
en=lLJSLKDSJADSKJJXNAJNDJKAH&version=1.0&id=19e02ej10d9j
92id
```

Methods: Query Parameters

✅	❌
• Clean base URL • Default to latest	• URL clutter, less discoverable

```
https://api.example.com/resources?&name=hehe&age=omg&token=1LJSLKDSJADSKJJXNAJNDJKAH&version=1.0&id=19e02ej10d9j92id
```

Methods: Header Versioning

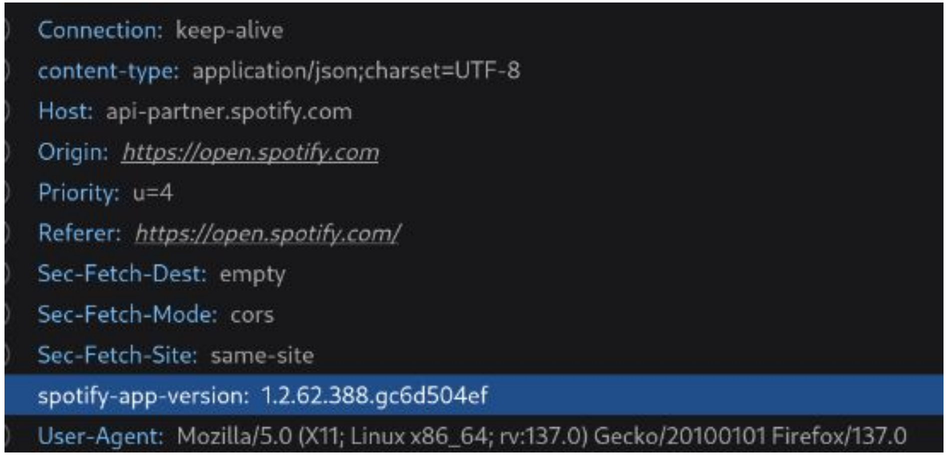
`Accept-Version: 1.0.1`

or

`X-API-Version: 1.0.1`

or

X-Custom-Name: `1.0.1`



```
Connection: keep-alive
content-type: application/json;charset=UTF-8
Host: api-partner.spotify.com
Origin: https://open.spotify.com
Priority: u=4
Referer: https://open.spotify.com/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site
spotify-app-version: 1.2.62.388.gc6d504ef
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:137.0) Gecko/20100101 Firefox/137.0
```

Methods: Header Versioning

✅	❌
● Clean URL	● Hard to test on browser ● Require extra API setup

Codebase organisation

Conclusion

- ***Plan early***
- ***Design for long-term growth***

Content

- Best practice
- Documentation
- Versioning
- Security

Best Practice: API Security

Authentication - Who Are You?

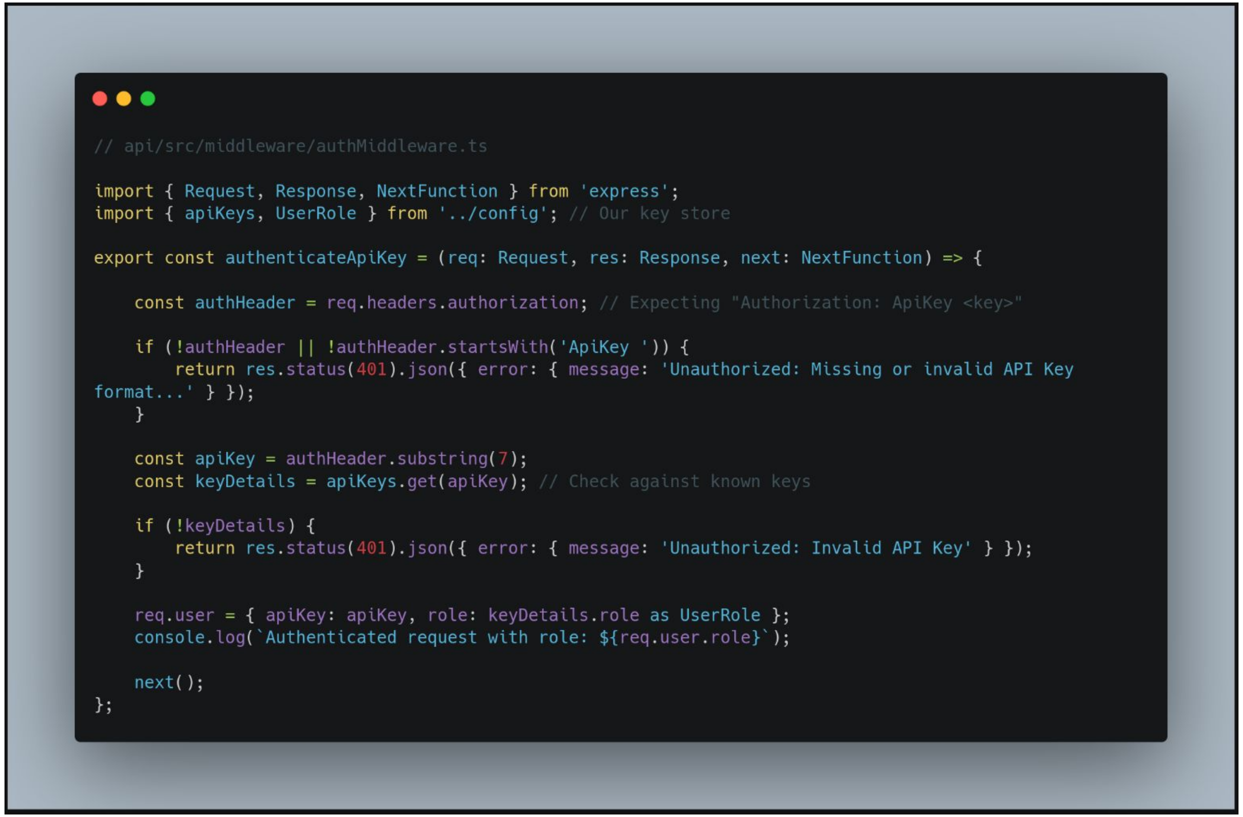
Why? Ensures only known entities can interact with your API. The first line of defense.

Common Methods:

- API Keys (Simple, good for service-to-service)
- OAuth 2.0 / OIDC (User delegation, complex scenarios)
- JWT (JSON Web Tokens) (Stateless sessions)
- Basic Auth (Simple, use only with HTTPS)

Best Practice: API Security

Authentication - Who Are You?

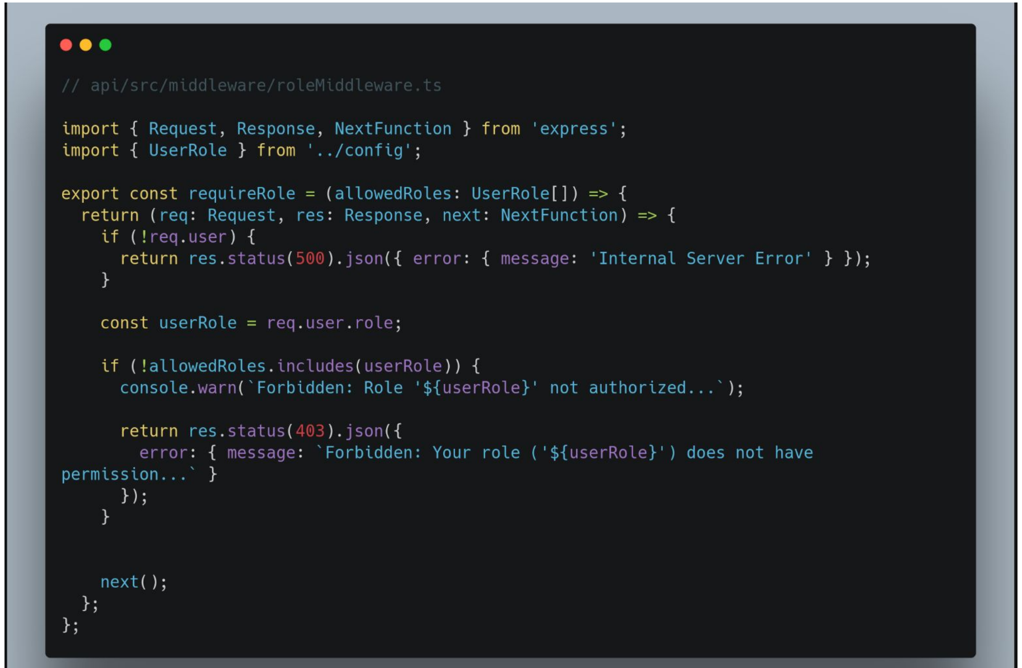


```
// api/src/middleware/authMiddleware.ts

import { Request, Response, NextFunction } from 'express';
import { apiKeys, UserRole } from '../config'; // Our key store

export const authenticateApiKey = (req: Request, res: Response, next: NextFunction) => {

    const authHeader = req.headers.authorization; // Expecting "Authorization: ApiKey <key>"

    if (!authHeader || !authHeader.startsWith('ApiKey ')) {
        return res.status(401).json({ error: { message: 'Unauthorized: Missing or invalid API Key
format...' } });
    }

    const apiKey = authHeader.substring(7);
    const keyDetails = apiKeys.get(apiKey); // Check against known keys

    if (!keyDetails) {
        return res.status(401).json({ error: { message: 'Unauthorized: Invalid API Key' } });
    }

    req.user = { apiKey: apiKey, role: keyDetails.role as UserRole };
    console.log(`Authenticated request with role: ${req.user.role}`);

    next();
};
```

Best Practice: API Security

Authorization - What Can You Do?

Why? Principle of Least Privilege. Users/services should only access what they need. Prevents privilege escalation.



```
// api/src/middleware/roleMiddleware.ts

import { Request, Response, NextFunction } from 'express';
import { UserRole } from '../config';

export const requireRole = (allowedRoles: UserRole[]) => {
  return (req: Request, res: Response, next: NextFunction) => {
    if (!req.user) {
      return res.status(500).json({ error: { message: 'Internal Server Error' } });
    }

    const userRole = req.user.role;

    if (!allowedRoles.includes(userRole)) {
      console.warn(`Forbidden: Role '${userRole}' not authorized...`);

      return res.status(403).json({
        error: { message: `Forbidden: Your role ('${userRole}') does not have
permission...` }
      });
    }


    next();
  };
};
```

Best Practice: API Security

Input Validation - Trust No One!

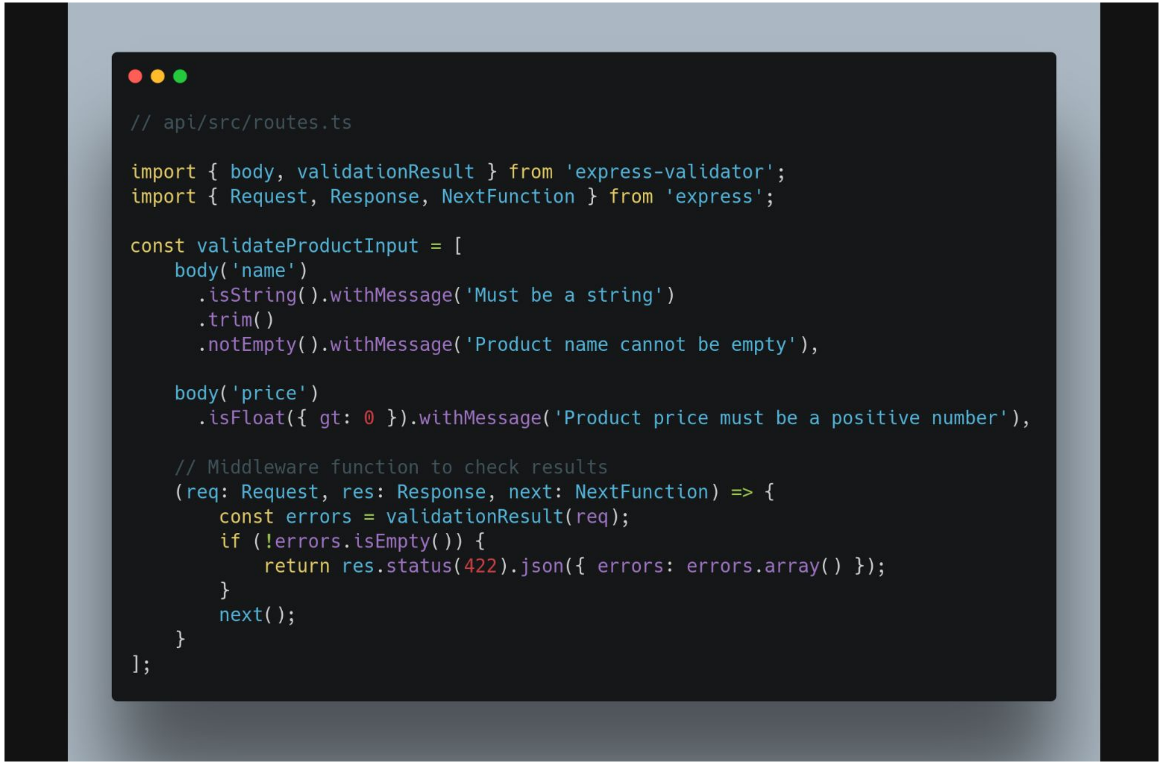
Why?

- Prevents data corruption / unexpected application states.
- Mitigates injection attacks (SQLi, NoSQLi, Command Injection, XSS - where applicable).
- Stops crashes caused by malformed data.
- Enforces data consistency.

What to Validate: Type, Format (regex), Length, Range, Allowed Values, Sanitize/Escape (if reflecting data).

Best Practice: API Security

Input Validation - Trust No One!



```ts
// api/src/routes.ts

import { body, validationResult } from 'express-validator';
import { Request, Response, NextFunction } from 'express';

const validateProductInput = [
    body('name')
      .isString().withMessage('Must be a string')
      .trim()
      .notEmpty().withMessage('Product name cannot be empty'),

    body('price')
      .isFloat({ gt: 0 }).withMessage('Product price must be a positive number'),

    // Middleware function to check results
    (req: Request, res: Response, next: NextFunction) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(422).json({ errors: errors.array() });
        }
        next();
    }
];
```

Best Practice: API Security

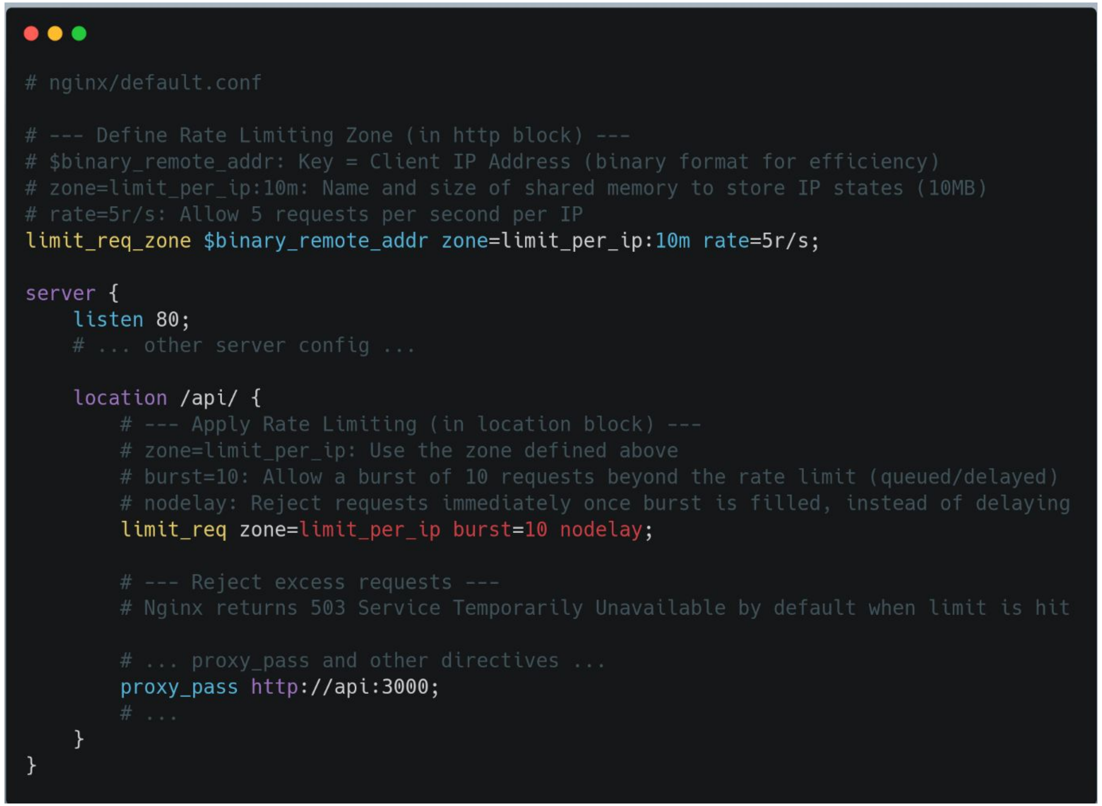
Rate Limiting - Preventing Abuse

Why?

- Protects against Denial-of-Service (DoS/DDoS) attacks.
- Prevents resource exhaustion (CPU, memory, database connections).
- Ensures fair usage among clients.
- Can help mitigate brute-force attacks on login endpoints.

Best Practice: API Security

Rate Limiting - Preventing Abuse



```
# nginx/default.conf

# --- Define Rate Limiting Zone (in http block) ---
# $binary_remote_addr: Key = Client IP Address (binary format for efficiency)
# zone=limit_per_ip:10m: Name and size of shared memory to store IP states (10MB)
# rate=5r/s: Allow 5 requests per second per IP
limit_req_zone $binary_remote_addr zone=limit_per_ip:10m rate=5r/s;

server {
    listen 80;
    # ... other server config ...

    location /api/ {
        # --- Apply Rate Limiting (in location block) ---
        # zone=limit_per_ip: Use the zone defined above
        # burst=10: Allow a burst of 10 requests beyond the rate limit (queued/delayed)
        # nodelay: Reject requests immediately once burst is filled, instead of delaying
        limit_req zone=limit_per_ip burst=10 nodelay;

        # --- Reject excess requests ---
        # Nginx returns 503 Service Temporarily Unavailable by default when limit is hit

        # ... proxy_pass and other directives ...
        proxy_pass http://api:3000;
        # ...
    }
}
```

Best Practice: API Security

Key Takeaways

Layer Your Security: No single technique is foolproof. Combine multiple defenses.

Authenticate Every Request: Know *who* is calling your API (API Keys, OAuth, JWT).

Authorize Every Action: Ensure the authenticated user has *permission* (RBAC, ABAC).

Validate ALL Input: Never trust data from the client. Define schemas/rules.

Implement Rate Limiting: Protect against DoS and abuse.

Use HTTPS Always: Encrypt data in transit (TLS 1.2+).

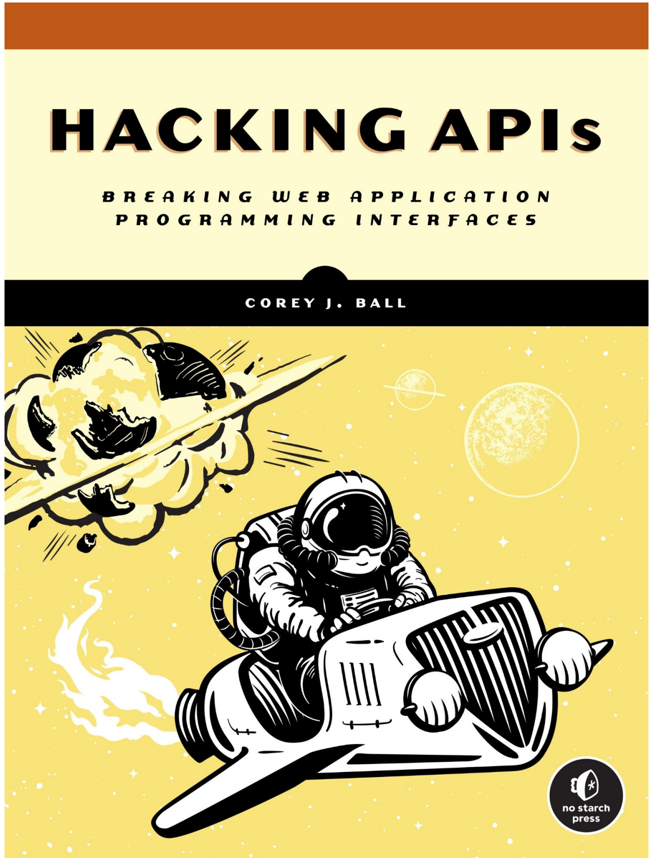
Secure Secrets: Don't hardcode keys/passwords. Use environment variables, secrets managers (Vault, AWS/GCP/Azure Secrets Manager).

Monitor & Log: Track API usage, errors, and security events.

Keep Dependencies Updated: Patch vulnerabilities in libraries/frameworks.

API Security

What should you read?



In real world…

Trello:
https://solsys.ca/lessons-from-trellos-api-exposure-securing-your-api-endpoints/

Twitter: https://www.localdefencebrisbane.org/blog/case-study-twitter-in-2022

Starbucks: https://www.facebook.com/share/p/1C2QuCBxwg/