API DESIGN

Contents

[Basics 5](#_Toc193657678)

[What are APIs 5](#_Toc193657679)

[HTTP 5](#_Toc193657680)

[HTTP Methods 5](#_Toc193657681)

[HTTP Header Fields 6](#_Toc193657682)

[HTTP Status Codes 7](#_Toc193657683)

[Cookies 7](#_Toc193657684)

[HTTP versions 9](#_Toc193657685)

[CORS 9](#_Toc193657686)

[HTTP Caching in API Design 10](#_Toc193657687)

[URL, Query & Path Parameters 11](#_Toc193657688)

[Content Negotiation 12](#_Toc193657689)

[Understand TCP / IP 12](#_Toc193657690)

[Basics of DNS 12](#_Toc193657691)

[Different API styles 15](#_Toc193657692)

[RESTful APIs 15](#_Toc193657693)

[Simple JSON APIs 18](#_Toc193657694)

[SOAP APIs 18](#_Toc193657695)

[GraphQL APIs 19](#_Toc193657696)

[gRPC APIs 19](#_Toc193657697)

[Building JSON / RESTful APIs 20](#_Toc193657698)

[How to make a REST API 20](#_Toc193657699)

[REST Principles 21](#_Toc193657700)

[URI Design 21](#_Toc193657701)

[Principles 21](#_Toc193657702)

[Versioning Strategies 23](#_Toc193657703)

[Handling CRUD Operations 23](#_Toc193657704)

[Understanding what a CRUD API is 23](#_Toc193657705)

[Best practices 24](#_Toc193657706)

[Pagination 25](#_Toc193657707)

[Rate Limiting 28](#_Toc193657708)

[Idempotency 29](#_Toc193657709)

[Practical advice on designing idempotent HTTP operations 30](#_Toc193657710)

[HATEOAS 30](#_Toc193657711)

[Error Handling 31](#_Toc193657712)

[RFC 7807 – Problem Details for APIs 31](#_Toc193657713)

[Authentication Methods 32](#_Toc193657714)

[Basic Authentication 32](#_Toc193657715)

[Token Based Authentication 32](#_Toc193657716)

[JWT 32](#_Toc193657717)

[OAuth 2.0 32](#_Toc193657718)

[Session Based Authentication 32](#_Toc193657719)

[Authorization Methods 33](#_Toc193657720)

[Role Based Access Control (RBAC) 33](#_Toc193657721)

[Attribute Based Access Control (ABAC) 33](#_Toc193657722)

[API Keys & Management 34](#_Toc193657723)

[API Documentation Tools 35](#_Toc193657724)

[Swagger / Open API 35](#_Toc193657725)

[Readme.com 35](#_Toc193657726)

[Stoplight 35](#_Toc193657727)

[Postman 35](#_Toc193657728)

[API Security 36](#_Toc193657729)

[Common Vulnerabilities 36](#_Toc193657730)

[Best Practices 36](#_Toc193657731)

[API Performance 37](#_Toc193657732)

[Performance Metrics 37](#_Toc193657733)

[Caching Strategies 37](#_Toc193657734)

[Load Balancing 37](#_Toc193657735)

[Rate Limiting / Throttling 37](#_Toc193657736)

[Profiling and Monitoring 37](#_Toc193657737)

[Performance Testing 37](#_Toc193657738)

[Error Handling / Retries 37](#_Toc193657739)

[API Integration Patterns 38](#_Toc193657740)

[Synchronous vs Asynchronous APIs 38](#_Toc193657741)

[Event Driven Architecture 38](#_Toc193657742)

[API Gateways 38](#_Toc193657743)

[Microservices Architecture 38](#_Toc193657744)

[Webhooks vs Polling 38](#_Toc193657745)

[Batch Processing 38](#_Toc193657746)

[Messaging Queues 38](#_Toc193657747)

[Rabbit MQ 38](#_Toc193657748)

[Kafka 38](#_Toc193657749)

[API Testing 39](#_Toc193657750)

[Mocking APIs 39](#_Toc193657751)

[Contract Testing 39](#_Toc193657752)

[Real-time APIs 40](#_Toc193657753)

[Web Sockets 40](#_Toc193657754)

[Server Sent Events 40](#_Toc193657755)

[API Lifecycle Management 41](#_Toc193657756)

[Standards and Compliance 42](#_Toc193657757)

[GDPR 42](#_Toc193657758)

[CCPA 42](#_Toc193657759)

[PCI DSS 42](#_Toc193657760)

[HIPAA 42](#_Toc193657761)

[PII 42](#_Toc193657762)

# Basics

Application Programming Interfaces (APIs) are a crucial part of today’s development. They allow software applications to communicate with each other and user functions from other software’s applications or services.

API design, therefore, becomes a key part of the software development process. The basics of API design encompass: (1) understanding the principles of what an API is, (2) types of APIs such as SOAP, REST and GraphQL and (3) understanding the standards and best practices in API design to ensure the development of powerful, user-friendly and secure APIs.

They define a common set of commands as well as the appropriate way to invoke these commands.

## What are APIs

Application Programming Interfaces abstract the complexity of applications to allow developers to use only the essentials of the software they are working with. They define the methods and data formats an application should use in order to perform tasks like sending, retrieving or modifying data. They allow applications to exchange data and functionality with ease, thus enabling integration and convergence of different technological services.

In other words, an API is a messenger that takes requests and tells a system what the user wants to do and replies back. Let’s think of API as a waiter. The user is the customer who is dinning at a restaurant and the service, the chef who will cook the food. However, we are missing one crucial component: how do the customer, who will order a certain dish, communicates it to the chef, who will cook it. Well, API will do just that. It is the link who will communicate the order to the chef, and deliver back the food to the customer.

## HTTPTTP

Hypertext Transfer Protocol, or HTTP, is a fundamental piece of any API as it is the protocol used for transmitting hypermedia data on the web, such HTML webpages or JSON, from a web API. Understanding HTTP means understanding requests and responses and how they should be constructed and handled. It will dictate how the APIs endpoints will be defined, how data should be transferred and what status codes should be used to convey certain specific scenarios. A solid grounding in HTTP will lead to a more robust, efficient and secure API.

Every we enter a website or make any kind of interactions, we are entering the cycle of requests and responses. Every single of those requests is independent. A website does not remember any specific requests. We can think of them as transactions. We could enhance the user experience through programming, local storage, cookies, sessions and some other tools, but at its core to avoid that, however, HTTP at its score is completely stateless.

Hypertext Transfer Protocol Secure, or HTTPS, is the same HTTP protocol with an additional layer of security to add using socket layers, SSL or TLS (Transport Security Layer). This is the standard for communication, as its secure. This can be done by installing a SSL certificate on the web.

### HTTP Methods

Each of the HTTP methods signifies a different type of request, allowing for a wide array of interactions with the API through endpoints.

There are several HTTP methods but the fundamental ones are:

* GET. Is it used when we want to get or load data from the web. It retrieves data from the server. Every time we visit a website, we are making a GET request to retrieve the HTML via HTTPS.
* POST. Used when we are posting data, adding a resource, to the server. For example, when we are submitting data via an HTML form. Theoretically we could also use a GET request for it, but it is way secure since the data is sent through the URL which is not secure at all.
* PUT. It will update data that is already on the server.
* DELETE. Just deletes data from the server.

### HTTP Header Fields

Every request and response using HTTP there is something called header and body.

The body typically, with the response, is the data we are trying to load from the server. When we make a request, it would be the data we are sending or submitting.

Headers play a crucial role as they provide essential information between client and server regarding the data to be exchanged. They can define parameters such as content type, authentication, response status, cookies and many more.

The header, on the other hand, has information related to the request and/or response. It has three different sections:

General

* Request URL.
* Request method.
* Status code.
* Remote address.
* Referrer policy.

Response

* Cookies.
* Accept-xxx.
* Content-Type.
* Content-Length.
* Authorization.
* User-Agent.
* Referrer.

Request

* Server. Usually hidden for security purposes.
* Set-Cookie.
* Content-Type.
* Content-Length.
* Date.

### HTTP Status Codes

Both requests and responses have what is called a status code. They usually go in ranges and they will let us know about the result of a request made to a server. They are 3-digit numbers where the first digit defines the class of response, while the last two do not have any categorization.

* **1xx:** informational. It means the request has been received and it is being processed.
* **2xx:** success. It has been received, understood and accepted.
  + 200 – OK.
  + 201 – OK created.
* **3xx:** redirect.
  + 301 – Moved to a new URL.
  + 304 – Not modified (cached version).
* **4xx:** client error. The request does not have what the server needs from the client. So, for example if the server needs a specific field from the sender and the body does not contain it, it will send back a 4xx error.
  + 400 – Bad request.
  + 401 – Unauthorized.
  + 404 – Not found.
* **5xx:** server error. The server has failed to receive and treat an apparently correct request.
  + 500 – Internal server error.

### Cookies

Cookies are small bits of data stored on a user’s browser that enables stateful HTTP sessions by storing pertinent information between server communications.

In API design, cookies are especially useful when authentication is required. They can store session tokens, allowing users to stay logged in across multiple sessions or different web pages.

They are sent from the web server to the client’s browser and then stored in the client’s device. They not only maintain the user’s session, but also can personalize their experience and enhance security.

API cookies

* They are randomly generated by the server, eliminating the need for the client to know the value.
* They have short expiration times.
* Primarily used for session-based authentication.

When consuming an API, it can be cumbersome to continuously enter username and password if required. API cookies simplify this process by allowing users to reuse credentials conveniently.

Advantages:

* Simplification of the authentication process.
* No need for clients to remember authentication information (save username and password).
* Ease of session management.

Disadvantages:

* Security risks.
* Potential cookie theft or tampering.
* Possibility of a sudden access loss due to expiration of the cookies.
* Server-side burden in managing cookies.

If an API cookie is stolen during the time period, they are valid, unauthorized access becomes a real possibility. Thus, using API cookies requires robust security measures and careful server-side management.

HTTP cookies

But API cookies are not the only kind of cookies that exists. So, what is the difference between API cookies and HTML cookies? First, similarities:

* They are both text data with attributes such as name, value and expiration.
* Data is included in HTTP request headers sent from client to server.
* The server can identify the client based on the cookie’s value.
* Once sent, the cookie information is stored on the client side.
* Session management is facilitated by managing expiration.

Differences are mainly based on their purpose, management and security measures:

* **Purpose.** API cookies are used for API authentication while HTTP cookies identify users on websites.
  + Diving more into the purpose of API cookies. (1) Session management: they help in managing the user’s sessions, storing identifies that enable a user to remain authorized as they navigate. (2) Personalization: they store the user’s preferences and settings, allowing web applications to tailor the user’s experience based on their choices. (3) Tracking and analytics: they can track the pages the user visits and the actions they take, data which can be then used for analytics and improving the user’s experience. And (4) security: they enhance security by storing tokens or flags that verify a user’s identity and permissions, helping prevent unauthorized access.
* **Issuer.** API cookies are sent by API servers, while HTTP are issued by web servers.
* **Expiration.** API cookies often have session-based expiration. HTTP cookies however, can have longer-term expiration times.
* **Security measures.** API cookies frequently use secret keys or tokens for added security (JWT) while HTTP cookies are often stored on the client side in plain text.
* **Management entity.** API cookies are primarily managed on the server side while HTTP cookies can be partly managed on the client’s side.

Types of cookies

Cookies come in various forms, each serving a different purpose and having a distinct lifespan. However, it is not set in store and a single cookie can sometimes belong to multiple types.

* Session cookies: temporary guests. They are stored temporarily for the duration of the browsing session and are then deleted as soon as the user closes the web browser.
* Persistent cookies: long-term residents. They stick around even after the user closes the web browser. They can stay from the range of minutes to several months, depending on their expiration date.
* Tracking cookies: digital private eyes.
* Authentication cookies: digital doormen.
* First-party and third-party cookies: insiders and outsiders. First-party cookies originate from the same domain as the website the user is visiting. Third-party cookies, on the other hand, are set by domains other than the one the user is currently visiting, often for tracking and advertising purposes.

Best practices for API cookies

To ensure you are using API cookies effectively and securely, some good practices are:

* Always secure sensitive data. Avoid storing sensitive information such as passwords in cookies. They should be used instead to store tokens or session identifiers.
* Regularly rotate tokens.
* Provide clear privacy policies. Inform the user of the usage of cookies and their purpose. Transparency is essential for compliance with data protection regulations.
* Regularly audit cookies. Periodically review and audit the cookies used in the application to ensure they align with the best practices and comply with regulations.
* Consider privacy regulations.

### HTTP versions

HTTP versions specify how data should be packaged and transported, as well as how web servers and browser should respond to commands.

For the most part we are using HTTP version 1.1 or HTTP/1.1. However, there’s now an upgraded version of it called HTTP/2. It is a major revision of the HTTP protocol that enhances things such as: (1) responds with more data, (2) reduced latency, (3) faster, more efficient and more secure. However, those changes are for the most part under the hood changes, meaning we do not have to worry about them.

Finally, there’s also a newer version, HTTP/3.

Each version basically brings improvements in speed, data transmission capabilities and security.

### CORS

Cross-origin resource sharing, or CORS, is a mechanism that uses HTTP headers to tell browser to give a web application running at one origin, access to selected resources from a different origin. So, if the user is in URL A, using CORS they could make a request to URL B.

By default, web browser prohibits making requests to a different domain than the one the web page came from (same-origin policy). CORS is a guideline that lets you configure a set of rules on the server to define which types of cross-domain requests are allowed, providing flexibility without compromising security.

For example, we could try to use an image that is hosted in a different website in ours. We could end up with a broken image, since the user, upon requesting the website, would not only request our site, but also the site that is hosting the image we are trying to load. This could be applied to even API requests.

One of the fields inside a header is the origin field. It contains from where the request has been originated. If the request goes to a server with the same origin, it would be allowed and then a 200 OK response would be received.

However, if that request goes through a different URL, it is called a cross-origin request. In this case, the header would contain the *Access-Control-Allow-Origin* to it. This field must match the origin field. If it a miss match, an error would be received. Furthermore, for security reasons, very little information about what went wrong is received. Only a CORS missing allow origin would be received.

The solution to a CORS error resides in the server-side of things. It must be configured with the correct header line or field. An *Access-Control-Allow-Origin* field must be adding, determining which origins can be accessed. If we are having a CORS error, we must configure the server to add this field to every single header response.

Certain HTTP request such as PUT, PATCH or DELETE or any other request with non-standard headers must be **preflight**. It is a sanity check **(preflight sanity check)** that checks if a request is safe to flight on the server. The browser automatically knows when the request has to preflight, and will make before a request to the server using the OPTIONS HTTP verb. The server will then respond to this OPTIONS request by saying which origins are allowed and also what methods are allowed (*Access-Control-Allow-Origin* and *Access-Control-Allow-Methods*) inside the header. Then, the main request, like a PUT, can be made without fear of being rejected. The server can also respond with an access-control-max-age, a field inside the header that allows the browser to preflight for a certain amount of time. Once that period of time has passed, the browser must make another OPTIONS request to obtain once again permissions from the server.

### HTTP Caching in API Design

HTTP caching involves storing copies of responses to HTTP requests to speed up future requests.

When an API receives the same request multiple times, instead of processing each request separately, it can use a previously stored response, thereby improving performance and efficiency.

The cache is governed by headers on the HTTP requests and response.

Applying a good HTTP caching can lead to drastically reducing latency, network traffic and improve the speed of an API.

The purpose of caching is to serve subsequent requests for the same data or resource more efficiently by avoided the need to recompute or retrieve the information from the source. It is applied at various levels of the application stack, including client-side (browser caching), server-side (server caching) and even intermediate points in the network (like content delivery networks or CDNs).

When a request is made for a resource, the caching mechanism first checks if the requested data is already available in cache. If it is present and hasn’t expired, the cached version is served. Else, the system must then fetch the data from the source.

Caching is a technique particularly effective for resources that are relatively static or don’t change frequently, such as images, stylesheets, scrips and certain API responses.

For REST APIs, being cacheable is one of its architectural constraints.

* GET requests should be cacheable by default – until a special condition arises. Usually, browser treat all GET requests as cacheable.
* POST requests are not cacheable by default, but can be made if either an *Expires* or *Cache-Control* header with a directive explicitly allows caching is added to the response. To indicate that a representation never expires, a service can include a time up to 1 year on the future as its expiration.  
  HTTP dates must always be expressed in GMT, never in local time.
* PUT and DELETE requests are not cacheable at all.

## URL, Query & Path Parameters

Uniform resource locators (URLs), query parameters and path parameters are components that play a crucial part in how API sends and retrieves data.

The URL forms the basis of the API given that it identifies the resource on the server that we are trying to access.

The query parameters are used to filter specific results, sorting or showing specific data fields.

On the other hand, path parameters serve as placeholders for variable data that will be input into the URL, allowing us to customize the data response.

Path and query parameters are both ways to tell the API what we are looking for in the server. But, which one should we use?

Query parameters

They are attached at the end of a URL and are, each, followed by a question mark.

* /persons**?name=John**&lastname=Doe

Query parameters act as filters. It will tell the server to filter the information in it by the parameters followed by a ?.

Path parameters

They are part of the URL itself and are enclosed in curly braces.

* /person/**{id}**

Path parameters are used when we are trying to identify a specific resource. While query parameters can return multiple resources, a path parameter should only return one. If done such way, it is considered a good RESTful practice.

Therefore…

***“When querying for multiple resources and we want to apply a filter, use a query parameter. On the other hand, when dealing with a single, specific item, use a path parameter.”***

[*https://medium.com/@averydcs/understanding-path-variables-and-query-parameters-in-http-requests-232248b71a8*](https://medium.com/@averydcs/understanding-path-variables-and-query-parameters-in-http-requests-232248b71a8)

## Content Negotiation

Content negotiation refers to the process where the client and the server communicate about the data representation which is acceptable for both of them.

It allows clients to indicate the preferred response format, such as JSON, XML or HTML.

This mechanism leads to flexible and adaptable APIS, enhancing usability. It **elevates** the **accessibility** of web APIs.

A specific document is often called resource. When a client wants to obtain a resource, the client will do so by a request via URL. The server then uses this URL to choose one of the variants available – each variant called representation – and returns a specific representation to the client.

The overall resource, as well as each representation, has a specific URL. Content negotiation will determine how a specific representation is chosen. It is identified by one of two mechanisms:

1. Specific HTTP headers by the client. This is called **server-driven negotiation** or proactive negotiation. Is it the standard.
2. Through HTTP response codes by the server. It is called **agent-driven negotiation** or reactive negotiation. They are used as fallback mechanisms and include:
   1. 300 – multiple choices.
   2. 406 – not acceptable.
   3. 415 – unsupported media type.

[*https://softwaremill.com/content-negotiation-in-practice/*](https://softwaremill.com/content-negotiation-in-practice/)

## Understand TCP / IP

TCP/IP stand for Transmission Control Protocol / Internet Protocol. It is the suite of communications protocols used to connect hosts on the Internet.

It provides ordered, error-checked delivery of streams of bytes from a program on one computer to another program on another computer.

## Basics of DNS

DNS or Domain Name System play a fundamental role in the way internet – and by extension, APIs – function. It acts as the internet’s equivalent of a phone book. It interprets human-friendly hostnames into machine-readable IP addresses that APIs need for communication. It is part of the navigation and messaging flow in API design.

In essence, DNS acts as a mapper, from domain name to an IP.

* [www.fireship.io](http://www.fireship.io) (sub-domain | domain | top.level domain (TLD) 🡪 172.16.254.1

Every time we type a URL in the browser, a DNS query is made to get the IP address of the hostname we are providing. First, it will attempt to look in the local browser or operative system cache. If the cache is empty, meaning that website has not been accessed before, or it has expired, then the query goes to the next stop: the phonebook.

For this, a recursive resolver server is in action. The DNS recursive server is recursive because it needs to make multiple requests to other servers, starting with the root server. This root server will respond with the address of the top-level domain, or TLD *(.com, .es, .io…)*. Then the resolver makes a second request to the TLD server. The TLD will respond with the IP address of the authoritative name server *(fireship.io)*. This will then respond with the final source of truth, which contains the IP address that we will be accessing in order to retrieve the content of the website we are trying to visit.

This information is then not only sent to the client for its used, but also saved in cache for future use.

When we register a domain name, it is handled by a registrar. The registrar is accredited by the non-profit organization ICANN. The record is maintained by a registry operator, who stores the DNS settings and propagates them to other DNS servers around the world.

In summary:

1. Local cache lookup. The browser or OS first checks its local cache to see if it already knows the IP address for the hostname we are trying to access. If found, it is then served immediately.
2. Recursive resolver contacted. If there’s no cache, then the request is forwarded to the DNS recursive resolver. Typically operated by an ISP or third-party provider such as Google or Cloudflare.
3. Root server queried. The recursive resolved contacts one of the root DNS servers, which responds with the IP address of the TLD name server.
4. The TLD server is queried. Then, upon querying the TLD sever, an address of the authoritative name server is given for that specific domain.
5. Authoritative name server queried. Finally, the recursive resolver contacts the authoritative name server, which responds and actual A record – the IP address of the requested domain.
6. Response and caching. The IP address is finally returned to the server and cached locally.

When a domain is registered, the following roles and components are involved: registrar, registry operator and registrant.

* Registrar. It is a company authorized by ICANN *(Internet Corporation for Assigned Names and Numbers)* to sell domain names. Examples: GoDaddy.
* Registry operator. Maintains the database of all domain names under a specific TLD and propagates data globally.
* Registrant. The person or entity that registers and owns the domain. They are given access to configure their DNS zone file – a set of DNS records defining how the domain behaves.

As the registrant, we are assigned a zone file where we can configure domain settings of our domain. Every zone has:

* Start of authority record or SOA. It tells another DNS servers who is in charge of that specific domain.
* Address or A record. It maps a domain or sub-domain to the IPv4 address of its host.
* AAAA record. Instead of an IPv4, it maps a IPv6 address.
* Canonical name or CNAME. We could forward a domain to another domain in the internet instead of an IP address. For this, a canonical name is used.
* MX or mail exchange record. Defines the mail server responsible for handling emails for the domain.
* TXT or text records. To store arbitrary data which is often used by third parties to verify ownership of the domain.
* Name server or NS record. It is a URL that the internet can ping in order to find the domain’s IP address. Usually, there is two name servers for better reliability.

# Different API styles

API design isn’t a one-size-fits-all endeavour. APIs can be structured in various styles, each with its own unique characteristics, advantages and use cases.

Early identification of the appropriate API style is crucial in ensuring functional, efficient and seamless end-user experience.

Commonly uses styles are: REST, SOAP, GraphQL, gRPC, WebSocket and Webhook.

* SOAP. XML based. It is extensively used in financial services and payment gateways. Its main characteristics are security and reliability. If looking for a light-weight app, SOAP can be an overkill due to its complexity and verbose.
* RESTful. They are the internet’s backbone. They extremely popular and easy to implement. They use HTTP methods. Most of the web services we interact daily, use powerful RESTful API. However, if we want real-time data or operate with a highly connected data model, REST might not be the fit.
* GraphQL. It is not only an architecture style for APIs, but also a query language. It allows clients to ask for specific data as they need. This means no more over-fetching or under-fetching of data. We ask exactly what we need. This leads to a more efficient network communication and faster responses. Facebook developed GraphQL to deliver data to its user. Nowadays, it is also used by other large companies such as Shopify and GitHub. Its flexibility and efficiency, makes it a great choice for applications with complex data requirements. However, it has a steep learning curve and also requires more processing on the server-side due to its flexible querying capabilities.
* gRPC. Modern and high performance, uses Protocol Buffers. Favourite for Microsoft microservices and companies like Netflix use gRPC to handle inter-service communication. However, if dealing with browser clients, gRPC might pose challenges due to limited browser support.
* WebSocket. Key characteristics are its real-time, bidirectional and persistent connections. Perfect for live chat applications and real-time gaming where low latency data exchange is crucial. If the application does not require real-time data, WebSocket might be an unnecessary overhead.
* Webhook. It’s an event-driven API using HTTP callbacks and asynchronous operations. For instance, GitHub uses Webhook to notify other systems whenever a new commit is pushed. But, if synchronous communications or immediate response is needed, Webhook is not the best bet.

## RESTful APIs

Representational State Transfer APIs are a set of conventions for designing networked applications. They utilize HTTP methods to read, update and delete data. RESTful APIs provide a simple and standardized way to build web services that can be easily consumed by different clients.

The key constraints and principals that an API must follow in order to be REST, or restful, areR:

1. Uniform interface. This is what gives REST APIs their consistent and predictable structure. It means that no matter what resource we are working with (users, products, posts), the way we interact with them will follow the same pattern.  
   By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions. The following constraints must be in place to achieve a uniform REST interface:
   1. Identification of resources. The interface must uniquely identify each resource involved in the interaction between the client and the server. For example: GET /users/123 will fetch user with ID 123. GET /products/10 fetches product with ID 10.
   2. Manipulation of resources through representations. The resources should have uniform representations in the server response. API consumers should use these representations to modify the resource state in the server. So, for example if we wanted to modify a user name we would send a JSON, the representation of the user, with the new name.
   3. Self-descriptive messages. Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource. Similar to the response header of an HTTP request.
   4. Hypermedia as the engine of application state (*HATEOAS*). The client should have only the initial URL of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.  
      Basically the server tells the client what it can do next by providing links (hypermedia) in the response. In other words, the client does not need to hardcore the API paths or know them ahead of time. It discovers what to do dynamically through the server’s responses.
2. Client-server. The client-server design enforces the separation of concerns, which helps the client and the server components evolve independently. By separating the user interface concerns (the client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. While the client and the server evolve, we have to make sure that the interface/contract between them does not break.  
   Basically the frontend and the backend are completely separate and communicate only through the API. As a benefit, the frontend can change without affecting the backend, and vice versa.
3. Stateless. Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request. The server cannot take advantage of any previously stored context information on the server. For this reason, the client application must entirely keep the session state.  
   If we require tokens in order to make the API call, the server must not remember anything about the previous requests. So, with each subsequent call, the token has to be provided.
4. Cacheable. It requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. If the response is cacheable, the client application gets the right to reuse the response later for equivalent requests and a specified period.
5. Layered system. The layered system allows an architecture to be composed of hierarchical layers by constraining component behaviour. In a layered system, each component cannot see beyond the immediate late they are interacting with. A layman’s example of a layered system is the MVC pattern.
6. Code on demand (optional). REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts. The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute that code.

This makes the API easy to understand, flexible and scalable. Moreover, it relieves heavily on the use of resources and their representations, making this design popular due to its performance, scalability, simplicity and reliability.

RESTful APIs are all about communication and restful. Restful web service is a service that uses web APIs to communicate.

They are…

* Simple and standardized. We do not have to worry about how to format data. It is standardized and industry used.
* Scalable and stateless. As the server grows in complexity, we can easily make modifications and the fact that they are stateless means that we do not have to worry about what data is in which state and keep track of that across client and server.
* High performance, in large part due to its support of cache.

The main building blocks of the RESTful API are the request that is sent to the server and the response that is sent from the server to the client.

What verbs or actions should we do? Basically, a CRUD, using HTTP methods or operations. Create (POST), read (GET), update (PUT) and delete (DELETE).

Request

The request should contain:

* Operation.
* Endpoint.
* Parameters or body. Data that is sent with the request.
* Headers. This is a special part that could contain some data such as authentication.

Response

The response is typically in the form of JSON data. Or sometimes XML.

Resource

They key abstraction of information in REST is a resource. Any information that we can name can be a resource. For example, a REST resource could be a document or image, a temporal service or a collection of other resources.

The state of the resource, at any particular time, is known as the resource representation. It consists of:

* The data.
* The metadata describing the data.
* Hypermedia links that help the clients transition to the next desired state.

REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

The data format of a representation is known as media type. The media type identifier a specification that defines how a representation is to be processed.

Resource representation should be self-descriptive. The client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource. So, in practice, we will create lots of custom media types – usually one media type associates with one resource.

Resource methods are used to perform the desired transition between two states of any resources. We should not wrongly associated resource methods to HTTP methods. For example, if we decide that the application API will use HTTP POST for updating a resource – instead of PUT – it’s all right. The application interface shall still be considered RESTful.

## Simple JSON APIs

Simple JavaScript Object Notation APIs are a popular form of API which utilise JSON to exchange data between servers and web applications.

This design gains prominence mainly for its simplicity, light weight and easy readability. In the context of API design, a well-structured JSON API allows developers to efficiently interact with the backend and retrieve only the data they need in a consistent and comprehensible manner.

JSON APIs defines how the client can ask for a resource to be fetch or modified, and how the server should respond to such request.

Let’s say we are fetching articles of a news site. Normally, we would get back a JSON with information about that article and maybe an id referencing the author. In JSON APIs, if we were to say GET /articles/?include=author, now the JSON we receive as a response must also contain the author of that article.

But we could also exclude certain information from the response. If we were to GET /articles?include=author&fields[people]=name, we would now obtain the article, the author of it and now of the field people, we would only obtain the name of the author, and no other possible unnecessary information such as age, address…

**We could say that a JSON APIs is a strict way of specification of how a JSON should look as a response.**

## SOAP APIs

Simple Object Access Protocol APIs or SOAP APIs are a standard communication protocol system that permits programs that run on different operating systems (OS) to communicate using HTTP and XML.

SOAP APIs offers a robust and well-defined process for interaction between various software applications, mostly over a network. They are highly extensible, versatile and support a wide range of communications protocols.

Despite being more complex compared to other APIs, SOAP APIs ensure high reliability and security, making them the choice for certain business-focused, high-transaction applications.

## GraphQL APIs

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for executing those queries with existing data.

Unlike REST, where you have predefined data return structures for each endpoint, GraphQL APIs are designed around a type system and enable the client application to precisely specify what data it needs from the server.

This gives a lot of flexibility and efficiency, leading to fewer round trips to the server and significantly enhancing the performance of the client application.

## gRPC APIs

gRPC is a platform agnostic serialization protocol that is used to communicate between services. It is a modern alternative to REST APIs.

It is a binary protocol that uses HTTP/2 as a transport layer. It is a high performance, open source, general-purpose RPC framework that puts mobile and HTTP/2 first.

Its main use case is for communication between two different languages within the same application. We could use Python to communicate with Java.

# Building JSON / RESTful APIs

Building JSON/RESTful APIs involves designing and implementing APIs that adhere to the architectural constraints of Representational State Transfer (REST).

These APIs use JSON as a format for information interchange due to its lightweight, easy-to-understand and universally accepted nature.

A well-designed RESTful API utilizing JSON is key in developing applications that are scalable, maintainable and easily integrated with other systems.

This design approach enables the resources on a server to be accessed and manipulated using standard HTTP protocols, facilitating the communication between services and systems. Furthermore, it enables client-server interactions to be stateless, meaning each request from a client must contain all the information needed by the server to understand and process the request.

RESTful APIs adhere to six principles, as seen before:

1. Uniform interface.
2. Client-server architecture.
3. Statelessness.
4. Cacheable.
5. Layered system.
6. Code on demand.

The process involves planning and identifying resources, designing intuitive URIs, choosing data formats (JSON), setting up the development environment, defining and implementing CRUD operations, ensuring security, testing and proper documentation. Adhering to best practices like endpoint naming, ensuring data security and graceful error handling, is crucial for building robust and efficient APIs.

REST APIs make data integration and ETL (extract, transform, load) processes more efficient and secure, allowing for easier data transfer between services.

* Efficiency in data transfer. They provide a standardized way to exchange data between different systems, ensuring that ETL processes can be done seamlessly.
* Real-time integration. They enable real-time data integration, ensuring timely insights and actions as data updates in one system are immediately reflected in another.
* Scalability. As data volumes grows, REST APIs can handle the increased load without significant changes to the existing infrastructure.
* Flexibility and interoperability. Being platform-agnostic, REST APIs ensure smooth communication between diverse systems, from legacy platform to modern cloud applications.

### How to make a REST API

It is more than writing code. It is about understanding the needs of the users, ensuring data security and designing a system that is both robust and scalable.

* Identify the resources – Object modelling. Being by pinpointing the core entities or resources for the API. These are the primary objects that we will be working with. For example, for a healthcare system those would be: patients, doctors and appointments.
* Create model URIs. Design intuitive URIs for each resource. For example: /patients for all patients, /patients/{id} for a specific patient and /doctors/{id}/patients for all patients associated with a specific doctor.
* Determine resource representations. Decide on the data format of the resources. JSON and XML are the most popular choices.
* Setting up the development environment. For instance, if using Python, Flask as a framework and Postman for testing.
* Defining endpoints. For each resource, CRUD operations must be defined. For example, for books:
  + POST /books. To add a new book.
  + GET /books. To get all books.
  + GET /books/{id}. To get a specific book.
  + PUT /books/{id}. To update a book’s details.
  + DELETE /books/{id}. To remove a book.
* Implement the CRUD operations. Develop the logic behind the endpoints, ensuring robust error handling.
* Setting up authentication and authorization. Implement security measures such as JWT to ensure only authorized users can access certain parts of the API.
* Testing. Test each endpoint. Consider automated tools for efficiency.
* Documentation. Properly documentation of the API using tools such Swagger.

## REST Principles

RESTful APIs adhere to [six principles](#REST_API_Principles):

1. Uniform interface.
2. Client-server architecture.
3. Statelessness.
4. Cacheable.
5. Layered system.
6. Code on demand.

## URI Design

URI or Uniform Resource Identifier is a string of characters used to identify a name or resource on the internet. Designing URIs carefully is a crucial part of creating a smooth API interface that is easy to understand, remember and use.

Good URI design ensures that related resources are grouped together in a logical manner and can greatly impact the usability and maintainability of an API. It involves crafting standardized, intuitive HTTP paths that take advantage of the hierarchical nature of URLs to provide a better structure to the API.

This hierarchy can then be used to expand the API over time without breaking the existing client’s functionality.

### Principles

Some of the general principes of URI design are the following ones.

A URI must represent an object, uniquely and permanently

They represent a data object on the Internet. It has to be unique, so it represents one URI per data object.

URIs should also be permanent. Meaning once created, we should not change it. Changing URIs often means breaking links, confusing users and even harming SEO. If we need to change a URI, the best way to do so is by setting up a redirection, a 301. This will redirect the user from the old URI to the new one. 301 response codes tell the browser, and search engines, that the content has been moved permanently while keeping traffic and search rankings the same.

Sometimes different URIs can lead to the same content. For example: /user, /user/, /USER. This should be avoided, as it confuses search engines. To help with this, we use canonical URL. Canonical URL are a special tag in webpages that tell the search engine which version of the URI is the official one. This will improve SEO and void duplicate content penalties.

Be as human-friendly as possible

This is the most fundamental factor behind URI design. URI should be designed with the end user in mind. SEO and ease of development should come second.

A general rule to keep it simple and user-friendly is to keep URI short and to the point. For example: /about is better than /about-acme-corp-page.

Another common mistake us relying on information that is not useful or important to the user. For example: /products/23 instead of /products/ballpoint-pen. While searching for ID is easier for the developer, the product number 23 tells nothing to the end user.

Consistency

Once a URI structure is picked, try to be consistent and follow it.

If needed, use 301 redirects.

“Hackable” URIs

Related to consistency. So, let’s say we have: /events/2010/01. By common sense we would expect that if we introduce /events/2009/01, we would get the previous year data of that event.

Keywords

URIs should be composed of keywords that are important to the content of the page. As a side effect, SEO is also improved.

Some other technical details to keep in mind are:

* No evidence of the underlying technology inside the URI. Meaning no .html, .aspx or anything else.
* No www. It violates the user-friendly rule. So mydomain.com should be the same as www.mydomain.com. Furthermore, if users type the second option, a redirect should be in place.
* Format. It should be something like: domain.com/[key information]/[name]/?[modifiers].
* All lowercase.
* Separated by hyphens (-).

## Versioning Strategies

As the APIs evolve over time to meet new business requirements and functionality enhancements, it is crucial to manage the changes in a way that doesn’t break the existing client applications.

This calls for effective versioning strategies in API design. There are different versioning strategies, like URI versioning, request header versioning and media type versioning, which are adopted based on ease of implementation, client compatibility and accessibility.

Let’s say we have an endpoint at /puppies. This will return data of the date of birth of puppies in the format of MM/DD/YYYY. However, later on we realise that we also need the time of birth of each puppies. So the format must change to HH:MM:SS DD/MM/YYYY. How do we add this new feature without causing problems for users that are already using the previous endpoint? We could do:

* /api/v1/puppies 🡪 version that formats the date as DD/MM/YYYY.
* /api/v2/puppies 🡪 new version that formats the date as HH:MM:SS DD/MM/YYYY.

However, this kind of versioning through URI could lead to increased length in the URI.

Another possible way to accomplish versioning is through headers.

* *Accept*: application/vnd.myapp.v1+json.

Like this we accomplish clean URLs and flexibility, but also gain in complexity and we must remember to include versioning in the header for every request.

## Handling CRUD Operations

When designing APIs, one needs to account for various types of interactions with data. These typically revolve around the CRUD operations.

Whether the API is designed for a banking app or a social media platform, they both will need to create, read or retrieve existing data, or update and delete.

Effective CRUD operations in API design facilitates seamless interaction between the front-end and back-end systems, and ensures proper data management, thereby improving user experience.

### Understanding what a CRUD API is

A CRUD API is an interface that allows client users to do CRUD operations on the server’s database resources. Therefore, CRUD operations for an API is nothing more than a flow of requests and responses from client to server, rather than your typical database statements.

RESTful APIs focus in architectural principles such as statelessness, resource-based interactions and uniform interface, but they do not dictate a specific data format. However, that does not mean that we should not pay attention to it. There are three points to consider when choosing a data format, whether it be for API requests or responses, data serialization and data deserialization:

1. Criteria specific to the business nature.
2. Developers experience.
3. Performance.

For API requests, information can be sent or requested using various methods, including URL parameters (/resource?key=1), HTTP status codes or methods, headers and request bodies (JSON). Even cookies can be used for this. The same goes for API responses.

Data serialization and deserialization is used to convert complex data structures such objects or collections into a linear format that can be easily transmitted over a network and then reconstructed once again on the receiving end. The common formats are JSON and XML. XML is usually preferred when interoperability with legacy or compliance with industry standards is required, since it provides a hierarchical structure and supports more complex data types and structures. There are other formats such as protocol buffers and JSONAPI.

### Best practices

Some include:

* CRUD naming conventions. Clean, organized CRUD naming convention is a must. We should document and share the information for proper implementation.
  + Singular or plural? Personal preference, but once chosen, keep to it.
  + Versioned resources.
  + Nested resources. /users/1/orders 🡪 all orders from user 1.
  + Use nouns, not verbs. /users instead of /getUsers.
* Versioning strategies. The API should be able to handle breaking changes through versioning strategies. For more information, [here](#Versioning_Strategies).
* Error handling and exceptions. *Gracefully failing*. Setting up specific error handling rather than very generic ones allows for faster troubleshooting.
* Authentication and authorization mechanisms. They are crucial, ensuring only authorized users or systems can access protected resources. They should be implemented from the beginning, as to avoid complex re-structuring of the API later on. Some authentication and authorization methods include:
  + Basic authentication. Includes sending username and password encoded in the HTTP request header. This is very basic, as we are sending credentials as plaintext, making it vulnerable to interception.
  + Token-based authentication. Involves issuing users a unique token upon successful authentication. The token is then included in subsequent requests as an authorization header or query parameter. The server validates the token and grants access only if it is valid and has not expired.
  + OAuth. It is an authorization framework that allows third-party applications to access a user’s resources on a server without exposing the user’s credentials.
  + JSON Web Tokens (JWT). It is a compact, self-contained token format that represent claims such as user credentials and permissions. It is also digitally signed to ensure integrity and authenticity. They are usually issued by an authentication server upon successful authentication and are included in subsequent requests as a bearer token.
* Rate limiting and throttling. As to prevent abuse and DoS attacks. Unlike authentication and authorization, it can be implemented later on without any issues.
* Use standard HTTP methods for each CRUD operation. This enhances clarity and predictability of the API.
* Use appropriate HTTP status codes.
* Keep the API consistent and predictable.
* Documentation for developers. Use OpenAPI, Swagger…
* Design the API around resources.
* Use HATEOAS.

[*https://www.forestadmin.com/blog/an-experts-guide-to-crud-apis-designing-a-robust-one/*](https://www.forestadmin.com/blog/an-experts-guide-to-crud-apis-designing-a-robust-one/)

## Pagination

Pagination provides a systematic approach to handling large amounts of data in a manageable way.

Instead of returning all data in a single response, which can be overwhelming and inefficient, APIs should implement pagination to deliver this data in smaller, more convenient parcels.

This allows for client applications to fetch data incrementally and only as needed, greatly enhancing performance and usability.

Pagination implementation can be done through different strategies, such as limit-offset, cursor-based or time-based pagination.

Offset and limit parameter

One way of achieving pagination is through indicating in the URL the limit. We can do so by limiting the amount of data we want to retrieve, but also setting a range. So, for example:

* /api/v1/users 🡪 returns all users.
* /api/v1/users?limit=5 🡪returns the first 5 users.
* /api/v1/users?limit=5&start=5 🡪 returns, starting at position 5, the next 5 users.

It is strongly advised to include information regarding the current limit, size, start, previous and next pages inside the response body. The current limit would be indicated as is, but the previous and next calls would be indicated inside the \_links part of the response. Finally, if we happen to be in the last page, a next link should not be included.

An example of a JSON response resource with the pagination metadata added:

|  |
| --- |
| {  "data": [...],  "pagination": {  "limit": 10,  "offset": 10,  "total": 100,  "\_links": {  "self": "/api/v1/users?limit=10&offset=10",  "previous": "/api/v1/users?limit=10&offset=0",  "next": "/api/v1/users?limit=10&offset=20"  }  }  } |

Cursor-based pagination

Instead of relying in numeric offsets, cursor-based pagination uses a unique identifier or token to mark the position in the dataset. This approach ensures stability when new data is added or existing data is modified. Hence, it is the best approach for real-time feeds or data that changes often.

The cursor can be based on various criteria, such as timestamp, primary key or encoded representation of the record.

Page-based pagination

It involves using a page parameter to specify the desired page number. The API consumer requests a specific page of data, and the API responds with the corresponding page, typically along with metadata such as the total number of pages or total record count.

This simplifies the navigation process but is also accompanied by other parameters such as limit.

* /api/v1/users?page=2&limit=10 🡪 retrieves the second page for users and limits the number of resources to 10.

It is a simple and user-friendly approach, especially for UIs with numbered pages.

|  |
| --- |
| {  "data": [...],  "pagination": {  "page": 3,  "limit": 20,  "total\_pages": 10,  "total\_items": 200,  "\_links": {  "self": "/api/v1/products?page=3&limit=20",  "previous": "/api/v1/products?page=2&limit=20",  "next": "/api/v1/products?page=4&limit=20"  }  }  } |

Time-based pagination

In scenarios where data has a temporal aspect to it, time-based pagination can be useful. It involves using time-related parameters such as *start\_time* and *end\_time*, to specify a time range for retrieving data.

Some best practices to keep in mind when using any kind of pagination techniques for APIs are:

* Use a common naming convention for pagination parameters. Adopt a consist naming convention, such as limit, page and size. Some standard parameter names are: limit, offset, page, cursor, start\_time, end\_time.
* Always include pagination metadata in API responses. Provide metadata related to the pagination state in the API response. This can include: total number of records, current page, total number of pages, links to the next and previous pages…
* Determine an appropriate page size. Select an optimal page size that balances the amount of data returned per page. Some guidelines are: page size 10-50 items and max limit 100.
* Allow configurable limits. For example, avoid abuses such as ?limit=1000, setting as max ?limit=100, for example.
* Consider implementing sorting and filtering options.
* Preserve pagination stability. Ensure that the pagination remains stable and consistent between requests: it must be deterministic. Newly added or deleted records should not affect the order or positioning of existing records during pagination. This ensures that the user can navigate data without encountering unexpected changes to it.
  + Use stable sorting mechanism.
  + Avoid changing data order.
  + Use unique and immutable identifiers.
  + Handle record deletion gracefully. If a resource is deleted between paginated requests, it should not affect the pagination order or cause missing records. For example, if record A is deleted, subsequent requests should not skip to record B without any explanation.
  + Use deterministic pagination techniques.
* Handle edge cases and error conditions. Account for edge cases such as reaching the end of the dataset, handling invalid or out-of-range page requests and gracefully handle the errors. Provide informative error messages and proper HTTP status codes to guide API consumers in handling pagination-related issues. Some things to consider regarding edge cases are:
  + Out-of-range page requests. Let’s say the user asks for /users?page=9999&limit=10. The user is expecting page number 9999 and 10 items of it, max. But what if we did not have that many pages? Then ideally, we would want to return a 200 OK, an empty result set but with the pagination information, avoiding a 404 or 500 response. This would look something like this:

|  |
| --- |
| {  "data": [],  "pagination": {  "page": 9999, # Page number requested  "limit": 10,  "total\_pages": 5, # This informs that we only have 5 pages  "total\_items": 200,  }  } |

* + Invalid pagination parameters. Return a 400 error with a clear message.
  + Handling empty result sets. Should return a 200 OK with an empty data array and correct metadata.
  + Server errors and exception handling.
  + Rate limiting and throttling. 429 Too Many Requests. Should include a Retry-After header.
  + Clear and informative error messages.
  + Consistent error handling approach.
* Consider cache strategies. Implement caching mechanisms to store paginated data or metadata that does not frequently change. Caching can help improve performance by reducing the load on the server and reducing the response time for subsequent requests. Some strategies to consider are:
  + Page-level caching. Cache the entire paginated resource for each page. Meaning caching the data along with the pagination metadata. Suitable for relatively static data that does not change frequently.
  + Result set caching. Cache the result of a specific query or combination of query parameters. This is useful when the same query parameters are frequently used, and the result set remains relatively stable for a certain period. Cache the result set and serve it directly for subsequent requests with the same parameters. We would store both the full query, filters and the returned data.
  + Time-based caching. Set an expiration date for cache based on the expected freshness of the data.
  + Conditional caching. Meaning the server can respond with a 304 Not Modified status if the client’s cached version is still valid.
  + Reverse proxy caching. Reverse proxies can cache the API responses and serve them directly without forwarding the request to the backend API server. This offloads the caching responsibility from the application server and improves performance.

[*https://dev.to/pragativerma18/unlocking-the-power-of-api-pagination-best-practices-and-strategies-4b49*](https://dev.to/pragativerma18/unlocking-the-power-of-api-pagination-best-practices-and-strategies-4b49)

## Rate Limiting

Rate limiting

Rate limiting dictates the number of API calls a client can make within a specific timeframe. This helps in managing resource allocation, preventing abuse of the API and maintaining the overall health of the API system, whether it be intentional (DoS attacks) or accidental (like a misconfigured client).

For example, an API may respond with an HTTP 429 Too Many Requests status code, along with information about when the user can try again.

Proper rate limiting measures should be in place to ensure the API’s stability, thereby delivering a consistent and reliable service to all consumers.

It works primarily by setting a limit on the frequency of client requests, thereby preventing individual users from overloading the system.

Throttling

Throttling in programming refers to the act of deliberately slowing down the execution of an operation so that it happens only at a controlled rate. While often confused with rate limiting, throttling typically works at the client-side or inside the application logic rather than at the API gateway or backend infrastructure. In essence, throttling ensures that an action is performed at most once in a given time interval, regardless of how often the function is called.

A practical use case is in UI event handling. Suppose a user is scrolling a page and an event listener is attached to a scroll event that updates the UI or fetches data from an API. Without throttling, this event may fire dozens or hundreds of times per second, leading to performance issues. Throttling can limit that function to run, say, once every 100 milliseconds, improving both frontend performance and backend efficiency if the event triggers network calls.

In API design, server-side throttling can also occur as part of rate-limiting strategies, where the server allows some requests but forces them to be spaced out. Instead of immediately blocking a client, it might delay processing or queue up requests, applying backpressure gradually rather than cutting off access entirely.

Debouncing

Debouncing, on the other hand, is a technique used to ensure that a function is only executed after a certain period of inactivity. In contrast to throttling, which ensures a function executes regularly at a certain interval, debouncing waits until the rapid firing of events stops, and then triggers the action once. This is particularly useful when you want to limit operations that should only happen once the user has finished an interaction.

A common use case is in search inputs or auto-complete fields. If you call an API every time the user presses a key while typing a search term, you might end up making 10+ calls in a few seconds. Debouncing delays the API call until the user has stopped typing for a set amount of time—say, 300 milliseconds. This avoids unnecessary calls and improves user experience by returning results only when the user appears to be done typing.

In the context of APIs and application performance, rate limiting protects the backend by enforcing request ceilings per client; throttling smooths out high-frequency actions by limiting how often a function runs within a timeframe; and debouncing waits until rapid, repeated input has stopped before running a function once. Each of these techniques contributes to building robust, performant applications and APIs that can handle both intentional load and accidental misuse gracefully.

## Idempotency

Idempotency in API design refers to the concept where multiple identical requests have the same effect as a single request. This means that no matter how many times a client sends the same request to the server, the server’s state stays the same after the first request.

Designing API’s to be idempotent is essential for reliability, as it allows retries without side-effects, reduces complexity in distributed systems and provides better user experience in unstable network conditions.

It is usually applicable to PUT, DELETE and sometimes POST methods in RESTful APIs.

Here are some facts about idempotency:

* Idempotency is a property of operations or API requests that ensures repeating the operation multiple times produces the same result as executing it once.
* Safe methods are idempotent, but not all idempotent methods are safe.
* HTTP methods like GET, HEAD, PUT, DELETE, OPTIONS and TRACE are inherently idempotent, while POST and PATCH are generally non-idempotent.
* Understanding and leveraging the idempotent nature of HTTP methods helps create more consistent, reliable and predictable web applications and APIs.
* Most HTTP methods used in REST APIs are idempotent, except for POST, and following REST principles can ensure proper usage of idempotent methods.

Idempotency is important because resources may be called multiple times if the network is interrupted. In this scenario, non-idempotent operations can cause significant unintended side-effects by creating additional resources or changing them unexpectedly.

Image a scenario where an API facilitates monetary transactions between accounts. A user sends a request to transfer 100€ from Account A to Account B. Due to network latency or other factors, the request is unintentionally duplicated. Without idempotency, the API would process the same request two times, resulting in an unintended transfer of 200€.

Developers need to be careful when implementing methods that are not inherently idempotent like POST and PATCH to avoid data duplication and other unintended side effects.

Why is PUT inherently idempotent but PATCH is not?

PUT is considered to be idempotent because it replaces the entire resource with the exact data provided. Therefore, sending the same PUT request multiple times will end up exactly in the same state.

On the other hand, PATCH is not idempotent. PATCH only applies partial changes, and those changes might not behave the same if repeated.

### Practical advice on designing idempotent HTTP operations

1. Understand the HTTP methods. Recognize which HTTP methods are naturally idempotent (GET, PUT, DELETE…) and design the API operations accordingly.
2. Use idempotency keys for non-idempotent operations. For operations that are not inherently idempotent such as POST and PATCH, use idempotency keys. These keys allow the server to recognize repeated requests and handle them appropriately, ensuring the operation is performed only once.   
   Idempotency keys are unique identifier provided by the client in the request header. You can implement them like this:
   1. Generate unique keys. Clients should generate a unique idempotency key for reach request. This key should be included in the request header.  
      *Idempotency-Key:* […].
   2. Store and check keys. On the server side, store the outcome of each request using the idempotency key. Before processing a new request, check if the key has already been used. If it has, return the stored response instead of processing it again.
   3. Implement expiry for keys. Optionally, implement an expiry mechanism for idempotency keys to free up storage after a certain period of time.
3. Implement state violation. Before performing any operation, check the current state of the resource to determine if the operation has already been completed. This validation helps in preventing duplicate operations.
4. Return consistent responses. Ensure that the API returns the same response for repeated idempotent requests.

[*https://blog.dreamfactory.com/what-is-idempotency*](https://blog.dreamfactory.com/what-is-idempotency)

***Idempotency is about the end result, not whether the server technically “does” something each time.***

## HATEOAS

Hypertext As The Engine Of Application State (HATEOAS) is a key concept in the design of RESTful APIs. It implies that the API delivers data as well as information about variable interactions.

By utilizing hypermedia, it contributes to the self-descriptiveness and discoverability of the API. When correctly implemented, clients only need generic knowledge about hypermedia, not specific API semantics, which can significantly simplify client implementations and make APIs more flexible to changes.

Is it one of the core fundamentals of what it means to be RESTful. [Here](#HATEOAS) for more information. An example of an API RESTful response that applies HATEOAS would be:

|  |
| --- |
| {  "userId": 42,  "name": "Alice",  "email": "alice@example.com",  "\_links": {  "self": {"href": "/users/42"},  "orders": {"href": "/users/42/orders"},  "update": {"href": "/users/42", "method": "PUT"},  "delete": {"href": "/users/42", "method": "DELETE"}  }  } |

## Error Handling

Error handling ensures stability, usability and reliability of the API in production. APIs are designed to help communicate systems with each other. However, there can be instances where these systems might encounter exceptions or errors.

The process of predicting, catching and managing these errors occurrences is what we refer as to Error Handling.

It involves defining and implementing specific strategies to detect, manage and inform consumers of any exception or error that occurs while executing requests. Configuring it appropriately provides a more robust and seamless communication experience, enabling developers to debug and rectify issues more efficiently.

## RFC 7807 – Problem Details for APIs

# Authentication Methods

## Basic Authentication

## Token Based Authentication

## JWT

## OAuth 2.0

## Session Based Authentication

# Authorization Methods

## Role Based Access Control (RBAC)

## Attribute Based Access Control (ABAC)

# API Keys & Management

# API Documentation Tools

API Documentation Tools are instrumental in conveying the intricacies of API design to both technical and non-technical stakeholders.

These tools help in creating comprehensive, easy-to-understand and searchable documentation encompassing all elements of an API such as its functions, classes, return types, arguments, and more.

Thorough documentation is central in API design as it fosters seamless adoption, effective implementation and efficient troubleshooting of APIs.

## Swagger / Open API

Swagger, also known as OpenAPI, is a set of tools specifically used for designing, building and documenting RESTful Web services.

API developers heavily rely on it due to its incredible feature for designing APIs with clear and easy-to-understand approach. By utilizing the OpenAPI Specifications (OAS), developers an accurately define a RESTful API that can easily be used across various programming languages. This powerful universal language is a key component for effective and efficient API design.

## Readme.com

Readme.com is renowned for providing a collaborative platform for creating beautiful, dynamic and intuitive documentation. It’s a tool which aids developers in outlining clear, comprehensive documentation for their API interfaces.

The API documentation created with Readme.com is not just about the presentation of information, but enhances the reader’s understanding by making it interactive. This interactive approach encourages practical learning and offers insights into how the API will behave under different circumstances.

## Stoplight

Stoplight is an advanced tool that offers a comprehensive platform for technical teams to handle all aspects of API design.

Leveraging Spotlight, teams can design, document and develop APIs in a more collaborative and streamlined manner. It uses an OpenAPI specification and allows users to design APIs visually, making API development easier. With its ability to auto-generate API documentation, performing API mock testing and providing API management features, Spotlight plays a crucial role in adopting a design-first approach in API development.

## Postman

Postman is a popular tool in web development for designing, testing and managing APIs. As a collaborative platform, it simplifies each step of the API lifecycle and streamlines collaboration across teams.

In the context of API design, it can be employed to design and mock APIs, automate testing and observe responses in a user-friendly interface. API endpoints can be organized into collections also in Postman for a well-structured and organized API design process.

# API Security

## Common Vulnerabilities

## Best Practices

# API Performance

## Performance Metrics

## Caching Strategies

## Load Balancing

## Rate Limiting / Throttling

## Profiling and Monitoring

## Performance Testing

## Error Handling / Retries

# API Integration Patterns

## Synchronous vs Asynchronous APIs

## Event Driven Architecture

## API Gateways

## Microservices Architecture

## Webhooks vs Polling

## Batch Processing

## Messaging Queues

### Rabbit MQ

### Kafka

# API Testing

## Mocking APIs

## Contract Testing

# Real-time APIs

## Web Sockets

## Server Sent Events

# API Lifecycle Management

# Standards and Compliance

## GDPR

## CCPA

## PCI DSS

## HIPAA

## PII