

#API Design Goals

An API or application programmable interface is a software contract that defines the expectations and interactions of a piece of code exposed to external users. This includes the parameters, response, errors, and API name.

We discuss how to design an API and what it takes to make the design scalable, extensible, and easy to use. HTTP APIs are widely used in software systems. HTTP is a stateless protocol and systems often expose APIs using it.

Things to ponder:

- 1) Name of API — sensible and convey the ask
- 2) Parameters — define them properly
str/int/float/char ...
— take required params only

Trade-off

- i) optimize to give desired result
- ii) in case extending it to accommodate more fields/data in response
debate if the field is sensible to be a result with the given name.

Adjust parameters to limit internal API calls since I/O operations can be expensive

i.e. if we add one param and reduce 2 calls then we should.

correctness

optimization

3) Response - give necessary info only because it might be heavy on the network

4) Errors: ① Only necessary ones
② No error is bad practice

c.g. `getAdmin(str groupId)`

no check
for str
it should not
be part of
your code

say dB has
char limit of
10, fail it; no
need to specify to
the people/user about
'query failed ... char limit'
rather 'groupId large'

=> All these are design decisions.

Act as an API expert and I will debate over a scenario.

Say I am a API provider, the api is simple a getDetails API with string parameter id. And the db I have has a limit of 10 chars only. So

1. Should i create a check that the input parameter is string

My opinion I should not, because when we create the SLA we already tell the clients or consumer that it is string, so they should ensure to send string

2. Now the length of id should be less than or equal to 10. Here should I take care of that or client.

Some of the things that came to me

a. If i create a check then for every hit I will have to do an extra computation which is large scale is time consuming and memory utilization.

b. if i dont then the query will fail which means I already knew the query would eventually fail but I still allowed it to go to the db

c. ask the clients to create a check on their side, but it would mean i am vulnerable to the client, if they dont it means I will go all the way to the db just to get error

d. nobody puts a check because it is agreed upon in the SLA

And I think the discussion would end up like, how many clients and their request counts are following the SLA and how many are not

What is your opinion?

Checking if the input parameter is a string:

While it's true that the SLA (Service Level Agreement) can specify that clients should send a string, in practice, robust API design often includes server-side validation. This is because clients may not always adhere to the specifications for various reasons (e.g., errors in client code, misunderstanding of the API requirements, or changes in the client application that are not reflected in its API usage). By ensuring that the input is a string on the server side, you can provide more informative error messages and potentially prevent unintended behavior or security vulnerabilities. However, the cost of this check is generally minimal compared to the overall processing time of a request, especially for a simple type check.

Enforcing the length restriction of the ID:

This is a more nuanced issue. Let's consider your points:

- a. Extra computation for checking length: The computational overhead of checking the length of a string is typically very low, especially compared to the overhead of a database operation. This small upfront cost can prevent unnecessary database queries, which are usually much more resource-intensive.
- b. Allowing queries to fail: Allowing queries to fail due to known validation issues is generally not a good practice. It can lead to unnecessary load on the database and potentially mask other, more serious errors. Additionally, it can result in less clear error messages for the client.
- c. Relying on clients to validate: While clients should validate inputs to ensure they are sending appropriate requests, you cannot guarantee that all clients will implement validation correctly or at all. Client-side validation improves user experience and reduces unnecessary network traffic, but it should not be the only line of defense.
- d. No checks based on SLA agreement: Solely relying on the SLA and not performing any validation assumes that all clients will adhere to the agreement at all times. In reality, errors and oversights happen. Providing clear, immediate feedback when a request does not meet the API requirements helps developers identify and fix issues more quickly.

Conclusion:

Your API should ideally perform basic validation checks, including ensuring that inputs are of the correct type and meet any specified constraints, such as length. These checks serve multiple purposes:

They protect the integrity and performance of your system by preventing invalid requests from consuming unnecessary resources.

They improve security by reducing the attack surface for injection attacks or other malicious inputs.

They enhance the developer experience by providing immediate, clear feedback when a request is malformed, helping clients to correct their requests more efficiently.

Ultimately, the goal is to balance performance and robustness. Given the relatively low cost of basic validation checks and the significant benefits they provide in terms of system integrity, security, and usability, implementing server-side validation is generally considered a best practice in API design.

API design in practice

www.gobalform / chat-messaging / getAdmins / v1
Route model function version control
(the group of function)

A single API should not do all complex task

e.g. `setAdmin(List <Admin>, str groupId)`

say you don't have the group, and you design it in such a way that if group does not exist, it creates a group and adds all these admin.

This is a bad design.

{ NO SIDE EFFECT }

→ this design is weird, rather break the responsibilities.

Trying to do everything here

- Atomicity

say give 404 if group does not exist
and now client makes another API call
to create group.

Try to call right action
Do NOT Pollute the API.

Response side

if response is large break the response

(i) Pagination, on the client side.

breaks stateless maybe.

works with page size and offset.

(ii) Fragmentation

break the response and send
packets of data and in the end
an end packet.

Fragmentation is more to do with
network layer.

How consistent you want the data?

perfect
consistency
(slow and
costly)

partial
consistency
(using cache)

Service degradation → Send only
some/necessary data in response when
the output is very large.

e.g. all profiles → send only name and
id, but not photo.

Technical Definitions and Implementations:

Fragmentation (in the context of APIs):

Technical Definition: Fragmentation, in API terms, often refers to breaking down a large data payload into smaller, more manageable pieces, but it's less commonly used in this context than in databases or file systems. More precisely, it might refer to the division of data across different endpoints or services.

Technical Implementation: An API might offer multiple endpoints to access different pieces of a dataset or different kinds of related data, instead of a single endpoint that returns all data en masse. The client would need to make multiple requests to different endpoints to gather all the necessary data.

Example: An API for a social media platform might have separate endpoints for user profiles, user posts, and comments. To reconstruct a user's profile with all their posts and comments, a client application would need to make separate requests to each of these endpoints.

Pagination:

Technical Definition: Pagination in the context of APIs is the process of dividing the response data into discrete subsets that are returned in sequential "pages." It's a technique used to manage and navigate through large sets of data returned by an API call.

Technical Implementation: An API implementing pagination will typically accept query parameters such as page (or offset) and limit (or pageSize), which dictate the starting point of the data subset and the number of records to return, respectively. The API response might also include metadata indicating the presence of more data and how to retrieve it (e.g., links to the next and previous pages).

Example: An API that returns a list of books might use pagination to limit the response to 10 books at a time. If there are 100 books available, the client can request them in sets of 10 by incrementing the page number in subsequent requests.

Layman's Definitions and Examples:

Fragmentation (in the context of APIs):

Layman's Definition: Think of fragmentation like visiting different sections of a supermarket for specific items instead of getting everything in one section. Each section (endpoint) provides a specific type of item (data).

Layman's Example: If you're organizing a party and need to buy supplies, you might visit the bakery for bread, the produce section for vegetables, and the beverage aisle for drinks. Each trip targets a specific need, similar to how an API client might make requests to different endpoints for specific types of data.

Pagination:

Layman's Definition: Pagination is like reading through a catalog that's too thick to handle all at once. Instead of flipping through the entire catalog, you're given the option to view a certain number of items at a time, with easy ways to move to the next or previous set of items.

Layman's Example: Imagine you're at a restaurant looking through a menu that's several pages long. The menu is divided into sections (appetizers, mains, desserts), and each section is only a page or two, making it easier to decide what you want to order without being overwhelmed by choices.

In the context of APIs, pagination is a common and well-understood method for handling large datasets, providing a balance between efficiency and usability. Fragmentation, as it relates to distributing data across multiple endpoints or services, involves considerations of design and architecture more than data management per se.

In terms of fragmentation I read, break the response in multiple packets and send small packets and in the last an end packet to signal end of response.

ChatGPT

The concept you're describing aligns more closely with fragmentation in networking, especially as it pertains to the transmission of data over a network, rather than how the term is typically used in relation to APIs or databases. In networking, fragmentation is indeed about splitting data into smaller packets for transmission, ensuring that each packet conforms to the maximum transmission unit (MTU) size of the network path. However, when discussing APIs, especially HTTP-based web APIs, this specific type of fragmentation is generally handled at a lower level (by the TCP/IP stack) rather than by the API itself or its consumers.

Technical Definitions and Implementations Revisited:

Fragmentation (Networking Context):

Technical Definition: Fragmentation in networking involves dividing larger packets or datagrams into smaller, manageable packets to match the MTU of the network path being used. This process is necessary because different networks may have different MTU limits, and packets larger than the MTU are either fragmented (if allowed) or dropped.

Technical Implementation: The IP protocol handles fragmentation transparently to higher-level protocols and applications. When a packet is too large for the network segment it must traverse, it's divided into smaller fragments. Each fragment contains a portion of the original data and header information that allows the receiving end to reassemble the original packet. The fragments are then transmitted independently and reassembled at the destination.

Example: Sending a large image over a network might require the data to be split into multiple packets. If the image is 10,000 bytes but the MTU is 1,500 bytes, the image data will be fragmented into several packets, each no larger than 1,500 bytes, and then reassembled once they reach the destination.

When applied to API responses, while the term "fragmentation" isn't typically used in this context, the concept of breaking down data can still be somewhat relevant, particularly in streaming or chunked transfer encoding. In such cases, the data is sent in a series of chunks, and the receiver knows that the transfer is complete when it receives an ending chunk or marker.

Clarified Example in API Context:

If we were to adapt the networking concept of fragmentation to APIs, especially large responses, it would be akin to streaming or chunked responses:

API Context Example: An API might stream a large JSON response in parts, especially useful in HTTP/1.1 where "chunked" transfer encoding is used. The client starts processing the initial chunks of data as they arrive, without needing to wait for the entire response. This approach is beneficial for large datasets or files. The end of the response is indicated by a zero-length chunk, signaling that all data has been sent.

This mechanism allows clients to begin processing data before the entire response is received, improving efficiency for large data transfers. However, in practice, this approach's specifics are managed by the HTTP protocol and underlying network protocols, rather than requiring explicit handling by API developers or consumers.

The handling of pagination, especially in the context of web services and APIs, doesn't typically involve storing paginated data between requests, due to the stateless nature of HTTP. Instead, pagination is dynamically computed based on the request parameters and the underlying dataset. Here's how it generally works and how systems handle the "next page" mechanism:

Handling Pagination:

Underlying Data Store: The data that an API provides access to is stored in a persistent storage system, such as a database. This could be a relational database like PostgreSQL or MySQL, a NoSQL database like MongoDB, or any other form of persistent storage.

Request Processing: When the API receives a request that includes pagination parameters (e.g., page number and limit or pageSize), it uses these parameters to compute which subset of the data to retrieve from the database.

Dynamic Query Execution: The API constructs a database query that fetches only the requested "page" of data. For example, if a client requests page 2 with a limit of 10 items per page, the API calculates that it needs to skip the first 10 items (page 1) and return items 11 through 20.

SQL Example: In SQL, this might involve the LIMIT and OFFSET clauses. For instance, `SELECT * FROM table LIMIT 10 OFFSET 10` would skip the first 10 records and return the next 10.

Statelessness and the "Next Page" Link: Since HTTP and RESTful APIs are stateless, each request is independent, and servers do not remember previous requests. The "next page" link provided in the response is typically constructed based on the current request's parameters. It is a URL that includes the pagination parameters for the next set of data. The server doesn't store page state between requests; instead, it relies on clients to request the correct page by using these links.

How the "Next Page" Link Works:

The "next page" link is a URL that the client can use to request the next set of data. It includes the API endpoint and query parameters for pagination, specifying the page number and size for the next chunk of data. The API calculates this link by incrementing the current page number and keeping the same page size, then embedding these values into the query parameters of the provided URL.

When the client follows this link (i.e., makes a new request to the URL), the API interprets the pagination parameters from the URL, queries the database accordingly, and returns the next set of data.

Example:

If the initial request is to an endpoint like `GET /items?page=1&limit=10`, the API might include in its response a "next page" link like `/items?page=2&limit=10`. When the client makes a request to this "next page" link, the API calculates the offset as $1 * \text{limit} = 10$ (for page 2) and fetches the next 10 items from the database.