# Common Design Patterns

## Empty Responses

The standard `Delete` method **should** return `google.protobuf.Empty`, unless it is performing a "soft" delete, in which case the method **should** return the resource with its state updated to indicate the deletion in progress.

For custom methods, they **should** have their own `XxxResponse` messages even if they are empty, because it is very likely their functionality will grow over time and need to return additional data.

## Representing Ranges

Fields that represent ranges **should** use half-open intervals with naming convention `[start_xxx, end_xxx)`, such as `[start_key, end_key)` or `[start_time, end_time)`. Half-open interval semantics is commonly used by C++ STL library and Java standard library. APIs **should** avoid using other ways of representing ranges, such as `(index, count)`, or `[first, last]`.

## Resource Labels

In a resource-oriented API, the resource schema is defined by the API. To let the client attach small amount of simple metadata to the resources (for example, tagging a virtual machine resource as a database server), APIs **should** use the resource labels design pattern described in `google.api.LabelDescriptor`.

To do so, the API design **should** add a field `map<string, string> labels` to the resource definition.

```
ge Book {
ing name = 1;
<string, string> labels = 2;
```

## Long Running Operations

If an API method typically takes a long time to complete, it can be designed to return a Long Running Operation resource to the client, which the client can use to track the progress and receive the result. The Operation (https://github.com/googleapis/googleapis/blob/master/google/longrunning/operations.proto) defines a standard interface to work with long running operations. Individual APIs **must not** define their own interfaces for long running operations to avoid inconsistency.

The operation resource **must** be returned directly as the response message and any immediate consequence of the operation **should** be reflected in the API. For example, when creating a resource, that resource **should** appear in LIST and GET methods though the resource **should** indicate that it is not ready for use. When the operation is complete, the `Operation.response` field should contain the message that would have been returned directly, if the method was not long running.

An operation can provide information about its progress using the `Operation.metadata` field. An API **should** define a message for this metadata even if the initial implementation does not populate the `metadata` field.

## List Pagination

Listable collections **should** support pagination, even if results are typically small.

**Rationale:** If an API does not support pagination from the start, supporting it later is troublesome because adding pagination breaks the API's behavior. Clients that are unaware that the API now uses pagination could incorrectly assume that they received a complete result, when in fact they only received the first page.

To support pagination (returning list results in pages) in a `List` method, the API **shall**:

- define a `string` field `page_token` in the `List` method's request message. The client uses this field to request a specific page of the list results.
- define an `int32` field `page_size` in the `List` method's request message. Clients use this field to specify the maximum number of results to be returned by the server. The server **may** further constrain the maximum number of results returned in a single page. If the `page_size` is `0`, the server will decide the number of results to be returned.

- define a `string` field `next_page_token` in the `List` method's response message. This field represents the pagination token to retrieve the next page of results. If the value is `""`, it means no further results for the request.

To retrieve the next page of results, client **shall** pass the value of response's `next_page_token` in the subsequent `List` method call (in the request message's `page_token` field):

```
istBooks(ListBooksRequest) returns (ListBooksResponse);

ge ListBooksRequest {
ing parent = 1;
32 page_size = 2;
ing page_token = 3;


ge ListBooksResponse {
eated Book books = 1;
ing next_page_token = 2;
```

When clients pass in query parameters in addition to a page token, the service **must** fail the request if the query parameters are not consistent with the page token.

Page token contents **should** be a url-safe base64 encoded protocol buffer. This allows the contents to evolve without compatibility issues. If the page token contains potentially sensitive information, that information **should** be encrypted. Services **must** prevent tampering with page tokens from exposing unintended data through one of the following methods:

- require query parameters to be respecified on follow up requests.
- only reference server-side session state in the page token.
- encrypt and sign the query parameters in the page token and revalidate and reauthorize these parameters on every call.

An implementation of pagination **may** also provide the total count of items in an `int32` field named `total_size`.

## List Sub-Collections

Sometimes, an API needs to let a client `List/Search` across sub- collections. For example, the Library API has a collection of shelves, and each shelf has a collection of books, and a client wants to search for a book across all shelves. In such cases, it is recommended to use standard `List` on the sub-collection and specify the wildcard collection id `"-"` for the parent collection(s). For the Library API example, we can use the following REST API request:

```
ttps://library.googleapis.com/v1/shelves/-/books?filter=xxx
```

the reason to choose `"-"` instead of `"*"` is to avoid the need for URL escaping.

## Get Unique Resource From Sub-Collection

Sometimes, a resource within a sub-collection has an identifier that is unique within its parent collection(s). In this case, it may be useful to allow a `Get` to retrieve that resource without knowing which parent collection contains it. In such cases, it is recommended to use a standard `Get` on the resource and specify the wildcard collection id `"-"` for all parent collections within which the resource is unique. For example, in the Library API, we can use the following REST API request, if the book is unique among all books on all shelves:

```
ttps://library.googleapis.com/v1/shelves/-/books/{id}
```

The resource name in the response to this call **must** use the canonical name of the resource, with actual parent collection identifiers instead of `"-"` for each parent collection. For example, the request above should return a resource with a name like `shelves/shelf713/books/book8141`, not `shelves/-/books/book8141`.

## Sorting Order

If an API method lets client specify sorting order for list results, the request message **should** contain a field:

```
g order_by = ...;
```

The string value **should** follow SQL syntax: comma separated list of fields. For example: `"foo,bar"`. The default sorting order is ascending. To specify descending order for a field, a suffix `" desc"` **should** be appended to the field name. For example: `"foo desc,bar"`.

Redundant space characters in the syntax are insignificant. `"foo,bar desc"` and `"  foo ,  bar  desc  "` are equivalent.

## Request Validation

If an API method has side effects and there is a need to validate the request without causing such side effects, the request message **should** contain a field:

```
validate_only = ...;
```

If this field is set to `true`, the server **must not** execute any side effects and only perform implementation-specific validation consistent with the full request.

If validation succeeds, `google.rpc.Code.OK` **must** be returned and any full request using the same request message **should not** return `google.rpc.Code.INVALID_ARGUMENT`. Note that the request may still fail due to other errors such as `google.rpc.Code.ALREADY_EXISTS` or because of race conditions.

## Request Duplication

For network APIs, idempotent API methods are highly preferred, because they can be safely retried after network failures. However, some API methods cannot easily be idempotent, such as creating a resource, and there is a need to avoid unnecessary duplication. For such use cases, the request message **should** contain a unique ID, like a UUID, which the server will use to detect duplication and make sure the request is only processed once.

```
unique request ID for server to detect duplicated requests.
is field **should** be named as `request_id`.
g request_id = ...;
```

If a duplicate request is detected, the server **should** return the response for the previously successful request, because the client most likely did not receive the previous response.

## Enum Default Value

Every enum definition **must** start with a `0` valued entry, which **shall** be used when an enum value is not explicitly specified. APIs **must** document how `0` values are handled.

If there is a common default behavior, then the enum value `0` **should** be used, and the API should document the expected behavior.

If there is no common default behavior, then the enum value `0` **should** be named as `ENUM_TYPE_UNSPECIFIED` and **should** be rejected with error `INVALID_ARGUMENT` when used.

```
Isolation {
Not specified.
LATION_UNSPECIFIED = 0;
Reads from a snapshot. Collisions occur if all reads and writes cannot be
logically serialized with concurrent transactions.
IALIZABLE = 1;
Reads from a snapshot. Collisions occur if concurrent transactions write
to the same rows.
PSHOT = 2;



en unspecified, the server will use an isolation level of SNAPSHOT or
tter.
tion level = 1;
```

An idiomatic name **may** be used for the `0` value. For example, `google.rpc.Code.OK` is the idiomatic way of specifying the absence of an error code. In this case, `OK` is semantically equivalent to `UNSPECIFIED` in the context of the enum type.

In cases where an intrinsically sensible and safe default exists, that value **may** be used for the '0' value. For example, `BASIC` is the '0' value in the Resource View (#resource_view) enum.

## Grammar Syntax

In API designs, it is often necessary to define simple grammars for certain data formats, such as acceptable text input. To provide a consistent developer experience across APIs and reduce learning curve, API designers **must** use the following variant of Extended Backus-Naur Form (EBNF) syntax to define such grammars:

```
ction  = name "=" [ Expression ] ";" ;
ssion  = Alternative { "|" Alternative } ;
native = Term { Term } ;
       = name | TOKEN | Group | Option | Repetition ;
       = "(" Expression ")" ;
n      = "[" Expression "]" ;
ition  = "{" Expression "}" ;
```

`TOKEN` represents terminal symbols defined outside the grammar.

## Integer Types

In API designs, unsigned integer types such as `uint32` and `fixed32` **should not** be used because some important programming languages and systems don't support them well, such as Java, JavaScript and OpenAPI. And they are more likely to cause overflow errors. Another issue is that different APIs are very likely to use mismatched signed and unsigned types for the same thing.

When signed integer types are used for things where the negative values are not meaningful, such as size or timeout, the value `-1` (and **only** `-1`) **may** be used to indicate special meaning, such as end of file (EOF), infinite timeout, unlimited quota limit, or unknown age. Such usages **must** be clearly documented to avoid confusion. API producers should also document the behavior of the implicit default value `0` if it is not very obvious.

## Partial Response

Sometimes an API client only needs a specific subset of data in the response message. To support such use cases, some API platforms provide native support for partial responses. Google API Platform supports it through response field mask. For any REST API call, there is an implicit system query parameter `$fields`, which is the JSON representation of a `google.protobuf.FieldMask` value. The response message will be filtered by the `$fields` before being sent back to the client. This logic is handled automatically for all API methods by the API Platform.

```
ttps://library.googleapis.com/v1/shelves?$fields=name
```

## Resource View

To reduce network traffic, it is sometimes useful to allow the client to limit which parts of the resource the server should return in its responses, returning a view of the resource instead of the full resource representation. The resource view support in an API is implemented by adding a parameter to the method request which allows the client to specify which view of the resource it wants to receive in the response.

The parameter:

- **should** be of an `enum` type
- **must** be named `view`

Each value of the enumeration defines which parts of the resource (which fields) will be returned in the server's response. Exactly what is returned for each `view` value is implementation-defined and **should** be specified in the API documentation.

```
ge google.example.library.v1;

ce Library {
 ListBooks(ListBooksRequest) returns (ListBooksResponse) {
ption (google.api.http) = {
 get: "/v1/{name=shelves/*}/books"
```

```
BookView {
Not specified, equivalent to BASIC.
K_VIEW_UNSPECIFIED = 0;

Server responses only include author, title, ISBN and unique book ID.
The default value.
IC = 1;

Full representation of the book is returned in server responses,
including contents of the book.
L = 2;


ge ListBooksRequest {
ing name = 1;

Specifies which parts of the book resource should be returned
in the response.
kView view = 2;
```

This construct will be mapped to URLs such as:

```
ttps://library.googleapis.com/v1/shelves/shelf1/books?view=BASIC
```

You can find out more about defining methods, requests, and responses in the Standard Methods (/apis/design/standard_methods) chapter of this Design Guide.

## ETags

An ETag is an opaque identifier allowing a client to make conditional requests. To support ETags, an API **should** include a string field `etag` in the resource definition, and its semantics **must** match the common usage of ETag. Normally, `etag` contains the fingerprint of the resource computed by the server. See Wikipedia (https://en.wikipedia.org/wiki/HTTP_ETag) and RFC 7232 (https://tools.ietf.org/html/rfc7232#section-2.3) for more details.

ETags can be either strongly or weakly validated, where weakly validated ETags are prefixed with `W/`. In this context, strong validation means that two resources bearing the same ETag have both byte-for-byte identical content and identical extra fields (ie, Content-Type). This means that strongly validated ETags permit for caching of partial responses to be assembled later.

Conversely, resources bearing the same weakly validated ETag value means that the representations are semantically equivalent, but not necessarily byte-for-byte identical, and therefore not suitable for response caching of byte-range requests.

For example:

```
is is a strong ETag, including the quotes.
3e4d5b6c7c"
is is a weak ETag, including the prefix and quotes.
2b3c4d5ef"
```

It's important to understand that the quotes really are part of the ETag value, and must be present in order to conform with RFC 7232 (https://tools.ietf.org/html/rfc7232#section-2.3). This means that JSON representations of ETags end up escaping the quotes. For example, the ETags would be represented in JSON resource bodies as:

```
rong
ag": "\"1a2f3e4d5b6c7c\"", "name": "...", ... }
ak
ag": "W/\"1a2b3c4d5ef\"", "name": "...", ... }
```

Summary of permitted characters in ETags:

- Printable ASCII only
    - Non-ASCII characters permitted by RFC 2732, but are less developer-friendly
- No spaces
- No double quotes other than in the positions shown above

- Avoid backslashes as recommended by RFC 7232 to prevent confusion over escaping

## Output Fields

APIs may want to distinguish between fields that are provided by the client as inputs and fields that are only returned by the server on output on a particular resource. For fields that are output only, the field attribute **shall** be documented.

Note that if output only fields are set in the request or included in a `google.protobuf.FieldMask`, the server **must** accept the request without error. The server **must** ignore the presence of output only fields and any indication of it. The reason for this recommendation is because clients often reuse resources returned by the server as another request input, e.g. a retrieved `Book` will be later reused in an UPDATE method. If output only fields are validated against, then this places extra work on the client to clear out output only fields.

```
ge Book {
ing name = 1;
Output only.
estamp create_time = 2;
```

## Singleton Resources

A singleton resource can be used when only a single instance of a resource exists within its parent resource (or within the API, if it has no parent).

The standard `Create` and `Delete` methods **must** be omitted for singleton resources; the singleton is implicitly created or deleted when its parent is created or deleted (and implicitly exists if it has no parent). The resource **must** be accessed using the standard `Get` and `Update` methods, as well as any custom methods that are appropriate for your use case.

For example, an API with `User` resources could expose per-user settings as a `Settings` singleton.

```
etSettings(GetSettingsRequest) returns (Settings) {
ion (google.api.http) = {
et: "/v1/{name=users/*/settings}"



pdateSettings(UpdateSettingsRequest) returns (Settings) {
ion (google.api.http) = {
atch: "/v1/{settings.name=users/*/settings}"
ody: "settings"





ge Settings {
ing name = 1;
Settings fields omitted.


ge GetSettingsRequest {
ing name = 1;


ge UpdateSettingsRequest {
tings settings = 1;
Field mask to support partial updates.
ldMask update_mask = 2;
```

## Streaming Half-Close

For any bi-directional or client-streaming APIs, the server **should** rely on the client-initiated half-close, as provided by the RPC system, to complete the client-side stream. There is no need to define an explicit completion message.

Any information that the client needs to send prior to the half-close **must** be defined as part of the request message.

## Domain-scoped names

A domain-scoped name is an entity name that is prefixed by a DNS domain name to prevent name collisions. It is a useful design pattern when different organizations define their entity names in a decentralized manner. The syntax resembles a URI without a scheme.

Domain-scoped names are widely used among Google APIs and Kubernetes APIs, such as:

- The Protobuf `Any` type representation: `type.googleapis.com/google.protobuf.Duration`
- Stackdriver metric types: `compute.googleapis.com/instance/cpu/utilization`
- Label keys: `cloud.googleapis.com/location`
- Kubernetes API versions: `networking.k8s.io/v1`
- The `kind` field in the `x-kubernetes-group-version-kind` OpenAPI extension.

## Bool vs. Enum vs. String

When designing an API method, it is very common to provide a set of choices for a specific feature, such as enabling tracing or disabling caching. The common way to achieve this is to introduce a request field of `bool`, `enum`, or `string` type. It is not always obvious what is the right type to use for a given use case. The recommended choice is as follows:

- Using `bool` type if we want to have a fixed design and intentionally don't want to extend the functionality. For example, `bool enable_tracing` or `bool enable_pretty_print`.
- Using an `enum` type if we want to have a flexible design but don't expect the design will change often. The rule of thumb is the enum definition will only change once a year or less often. For example, `enum TlsVersion` or `enum HttpVersion`.
- Using `string` type if we have an open ended design or the design can be changed frequently by an external standard. The supported values must be clearly documented. For example:
  - `string region_code` as defined by Unicode regions (http://www.unicode.org/reports/tr35/#unicode_region_subtag).
  - `string language_code` as defined by Unicode locales (http://www.unicode.org/reports/tr35/#Unicode_locale_identifier).

## Data Retention

When designing an API service, data retention is a critical aspect of service reliablity. It is common that user data is mistakenly deleted by software bugs or human errors. Without data retention and corresponding undelete functionality, a simple mistake can cause catastrophic business impact.

In general, we recommend the following data retention policy for API services:

- For user metadata, user settings, and other important information, there should be 30-day data retention. For example, monitoring metrics, project metadata, and service definitions.
- For large-volume user content, there should be 7-day data retention. For example, binary blobs and database tables.
- For transient state or expensive storage, there should be 1-day data retention if feasible. For example, memcache instances and Redis servers.

During the data retention window, the data can be undeleted without data loss. If it is expensive to offer data retention for free, a service can offer data retention as a paid option.

## Large Payloads

Networked APIs often depend on multiple network layers for their data path. Most network layers have hard limits on the request and response size. 32MB is a commonly used limit in many systems.

When designing an API method that handles payloads larger than 10MB, we should carefully choose the right strategy for usability and future growth. For Google APIs, we recommend to use either streaming or media upload/download to handle large payloads. With streaming, the server incrementally handles the large data synchronously, such as Cloud Spanner API. With media, the large data flows through a large storage system, such as Google Cloud Storage, and the server can handle the data asynchronously, such as Google Drive API.

Previous
← Naming Conventions (/apis/design/naming_convention)
Next

Documentation (/apis/design/documentation) →