REST API Standards

The document contains standards that were forked, then adapted for ICT, from *Microsoft Azure REST API Guidelines*

The format of API standards differ from ICT guideline for writing standards as a result of the format of the source material.

Introduction

These guidelines apply to **all products and teams** implementing both **external and internal APIs**. They offer prescriptive guidance that ICT service teams **MUST** follow ensuring that internal and external customers of ICT have a great experience by designing APIs meeting these goals:

- Developer friendly via consistent patterns & web standards (HTTP, REST, JSON)
- Efficient & cost-effective
- Work well with SDKs in many programming languages
- Customers can create fault-tolerant apps by supporting retries/idempotency/optimistic concurrency
- Sustainable & versionable via clear API contracts with 2 requirements:
  - Customer workloads must never break due to a service change
  - Customers can adopt a version without requiring code changes

Technology and software is constantly changing and evolving, and as such, this is intended to be a living document. Contact the API Stewardship Board to suggest a change or propose a new idea. Please read the *Considerations for Service Design* for an introduction to the topic of API design for ICT services. *For an existing GA'd service, don't change/break its existing API; instead, leverage these concepts for future APIs while prioritizing consistency within your existing service.*

Contract First API Development

In accordance with the Guiding Principle Adopt API First Design Approach an API-first development approach should be adopted for all Nimbus application. Additionally other ICT product family should adopt this approach where possible.

Contract First API development encourages API contracts be written first before writing any code. According to the OpenAPI specification a contract can be written in YAML or JSON file, and is both human and machine readable. Both clients and service implementations can be code generated from the API contract.

This approach can have some draw backs, including slowing the service implementation in some cases as development teams must spend time designing the interface before coding can begin in ernest. However this approach also brings many advantages such as reusability, clear separation of concerns, reduction in blocking between teams, consistent communication between teams, easy documentation, consistent API designs.

API Stewardship Board

When a new API is proposed for a product that will be deployed to the **Nimbus platform**, an API Stewardship Board review **must** be completed before any significant development begins. For **non-Nimbus projects**, the API Stewardship Board can be contacted to discuss any questions relating to the standards however a review is not required.

Prescriptive Guidance

This document offers prescriptive guidance labeled as follows:

> **API-EXAMPLE-001**
>
> **DO** adopt this pattern. When deploying to Nimbus if you feel you need an exception, contact the API Stewardship Board **prior** to implementation.

> **API-EXAMPLE-002**
>
> **YOU SHOULD** adopt this pattern. When deploying to Nimbus if not following this advice, you MUST disclose your reason during the API Stewardship Board review.

API-EXAMPLE-003

YOU MAY consider this pattern if appropriate to your situation. No notification to the ICT API Stewardship Board is required.

API-EXAMPLE-004

YOU SHOULD NOT adopt this pattern. When deploying to Nimbus if not following this advice, you MUST disclose your reason during the API Stewardship Board review.

API-EXAMPLE-005

DO NOT adopt this pattern. When deploying to Nimbus if you feel you need an exception, contact the API Stewardship Board **prior** to implementation.

If you feel you need an exception, or need clarity based on your situation, please contact the ICT API Stewardship Board ***prior*** *to release of your API.*

Contract-first API Development

Contract First API development encourages API contracts be written first before writing any code. According to the OpenAPI specification a contract can be written in YAML or JSON file, and is both human and machine readable. Both clients and service implementations can be code generated from the API contract.

This approach can have some draw backs, including slowing the service implementation in some cases as development teams must spend time designing the interface before coding can begin in ernest. However this approach also brings many advantages such as reusability, clear separation of concerns, reduction in blocking between teams, consistent communication between teams, easy documentation, consistent API designs.

API-CONTRACT-001

DO Create a OpenAPI document using the standards described in this document before beginning development. When deploying to Nimbus follow the guidance of the API Stewardship Board review process.

Building Blocks: HTTP, REST, & JSON

ICT products can expose its APIs through the core building blocks of the Internet; namely HTTP, REST, and JSON. This section provides you with a general understanding of how these technologies should be applied when creating your service.

HTTP

ICT Platform services must adhere to the HTTP specification, *RFC7231*. This section further refines and constrains how service implementors should apply the constructs defined in the HTTP specification. It is therefore, important that you have a firm understanding of the following concepts:

- *Uniform Resource Locators (URLs)*
- *HTTP Request / Response Pattern*
- *HTTP Query Parameters and Header Values*

Uniform Resource Locators (URLs)

A Uniform Resource Locator (URL) is how developers access the resources of your service. Ultimately, URLs are how developers form a cognitive model of your service's resources.

> **API-URL-001**
>
> **DO** use this URL pattern:
>
> ```
> https://<product-ict-domain\>/<service-root\>/<api-name\>/<version
> \>/<resource-collection\>/<resource-id\>
> ```

example

```
GET https://ictplatform.co.uk/amm/model-management/v1/models/13123
```

Where:

Field	Description
customer-name	Name of the ICT customer
product-ict-domain	The domain for the SaaS solution
service-root	Service Specific path – should equate to a product eg Radar, Igloo
api-name	Name of the API / Microservice
version	Major version number in the format v<n> eg v1
resource-collection	Name of the collection, unabbreviated, pluralized
resource-id	Id of resource within the resource-collection. This MUST be the raw string/number/guid value with no quoting but properly escaped to fit in a URL segment.

API-URL-002

DO use kebab-casing (preferred) or camel-casing for URL path segments. If the segment refers to a JSON field, use camel casing.

kebab-casing example

```
www.example.com/this-is/a-nice/url-name
```

API-URL-003

DO return *414-URI Too Long* if a URL exceeds 2083 characters.

API-URL-004

DO treat service-defined URL path segments as case-sensitive. If the passed-in case doesn't

match what the service expects, the request **MUST** fail with a *404-Not found* HTTP return code.

Some customer-provided path segment values may be compared case-insensitivity if the abstraction they represent is normally compared with case-insensitivity. For example, a UUID path segment of 'c55f6b35-05f6-42da-8321-2af5099bd2a2' should be treated identical to 'C55F6B35-05F6-42DA-8321-2AF5099BD2A2'.

API-URL-005

DO ensure proper casing when returning a URL in an HTTP response header value or inside a JSON response body.

API-URL-006

DO restrict the characters in service-defined path segments to *0-9 A-Z a-z - . _ ~*, with *:* allowed only as described below to designate an action operation.

API-URL-007

YOU SHOULD restrict the characters allowed in user-specified path segments (i.e. path parameters values) to *0-9 A-Z a-z - . _ ~* (do not allow :).

API-URL-008

YOU SHOULD keep URLs readable; if possible, avoid %-encoding (ex: Cádiz is %-encoded as C%C3%A1diz)

API-URL-009

YOU MAY use these other characters in the URL path but they will likely require %-encoding [*RFC 3986*]: */ ? # [] @ ! $ & ' ( ) * + , ; =*

API-URL-010

DO return URLs in response headers/bodies in a consistent form regardless of the URL used

to reach the resource. Either always a UUID for \<*tenant*> or always a single verified domain.

API-URL-011

YOU MAY use URLs as values

https://api.contoso.com/items?url=https://resources.contoso.com/shoes/fancy

API-URL-012

DO use UUID for entity ids

HTTP Request / Response Pattern

The HTTP Request / Response pattern dictates how your API behaves. For example: POST methods that create resources must be idempotent, GET method results may be cached, the If-Modified and ETag headers offer optimistic concurrency. The URL of a service, along with its request/response bodies, establishes the overall contract that developers have with your service. As a service provider, how you manage the overall request / response pattern should be one of the first implementation decisions you make.

Cloud applications embrace failure. Therefore, to enable customers to write fault-tolerant applications, *all* service operations (including POST) **must** be idempotent. Implementing services in an idempotent manner, with an "exactly once" semantic, enables developers to retry requests without the risk of unintended consequences.

Exactly Once Behavior = Client Retries & Service Idempotency

API-HTTP-001

DO ensure that *all* HTTP methods are idempotent.

API-HTTP-002

YOU SHOULD use PUT or PATCH to create a resource as these HTTP methods are easy to implement, allow the customer to name their own resource, and are idempotent.

API-HTTP-003

YOU MAY use POST to create a resource but you must make it idempotent and, of course, the response **MUST** return the URL of the created resource with a 201-Created. One way to make POST idempotent is to use the Repeatability-Request-ID & Repeatability-First-Sent headers (See *Repeatability of requests*).

HTTP Return Codes

Request Codes

Method	Description	Response Status Code
PATCH	Create/Modify the resource with JSON Merge Patch	*200-OK*, *201-Created*
PUT	Create/Replace the *whole* resource	*200-OK*, *201-Created*
POST	Create new resource (ID set by service)	*201-Created* with URL of created resource
POST	Action	*200-OK*
GET	Read (i.e. list) a resource collection	*200-OK*
GET	Read the resource	*200-OK*
DELETE	Remove the resource	*204-No Content*; avoid *404-Not Found*

API-HTTP-004

DO adhere to the ICT defined return codes (above) when the method completes synchronously and is successful.

API-HTTP-005

DO return status code *202-Accepted* and follow the guidance in *Long-Running Operations & Jobs* when a PUT, POST, or DELETE method completes asynchronously.

API-HTTP-006

DO treat method names as case sensitive and should always be in uppercase

API-HTTP-007

DO return the state of the resource after a PUT, PATCH, POST, or GET operation with a *200-OK* or *201-Created*.

API-HTTP-008

DO return the entity you just created for PUT and POST operations

API-HTTP-009

DO return a *204-No Content* without a resource/body for a DELETE operation (even if the URL identifies a resource that does not exist; do not return *404-Not Found*)

API-HTTP-010

DO return a *200-OK* from a POST Action. Include a body in the response, even if it has not properties, to allow properties to be added in the future if needed.

Response Codes

Method	Description	Response Status Code
GET	Path incorrect eg GET /{entity_name}/{id} when where entity_name does not exist	*404-Not-Found*

Method	Description	Response Status Code
GET	Entity if does not exist eg GET /{entity_name}/{id} when id does not exist	*404-Not-Found*
POST/PUT/PATCH /DELETE	Query parameter is incorrect	*404-Not-Found*
POST/PUT/PATCH	Badly formatted request body sent by the client	*400-Bad-Request*
ANY	Accessing any resource without appropriate authorization	*401-Unauthorized*
ANY	Unexpected error in the processing of the request	*500-Internal Server Error*

API-HTTP-011

DO adhere to the ICT defined return codes (above) when the method completes synchronously and is **not** successful:

API-HTTP-012

DO return a *403-Forbidden* when the user does not have access to the resource *unless* this would leak information about the existence of the resource that should not be revealed for security/privacy reasons, in which case the response should be *404-Not Found*. [Rationale: a *403-Forbidden* is easier to debug for customers, but should not be used if even admitting the existence of things could potentially leak customer secrets.]

API-HTTP-013

DO support caching and optimistic concurrency by honoring the the *If-Match*, *If-None-Match*, if-modified-since, and if-unmodified-since request headers and by returning the ETag and last-modified response headers

HTTP Query Parameters and Header Values

Because information in the service URL, as well as the request / response, are strings, there must be a predictable, well-defined scheme to convert strings to their corresponding values.

> **API-HTTP-014**
>
> **DO** validate all query parameter and request header values and fail the operation with *400-Bad Request* if any value fails validation. Return an error response as described in the *Handling Errors* section indicating what is wrong so customer can diagnose the issue and fix it themselves.

String translations

Data type	Document that string must be
Boolean	true / false (all lowercase)
Integer	-253+1 to +253-1 (for consistency with JSON limits on integers *RFC8259*)
Float	*IEEE-754 binary64*
String	(Un)quoted?, max length, legal characters, case-sensitive, multiple delimiter
UUID	123e4567-e89b-12d3-a456-426614174000 (no {}s, hyphens, case-insensitive) *RFC4122*
Date/Time (Header)	Sun, 06 Nov 1994 08:49:37 GMT *RFC7231, Section 7.1.1.1*
Date/Time (Query parameter)	YYYY-MM-DDTHH:mm:ss.sssZ (with at most 3 digits of fractional seconds) *RFC3339*
Byte array	Base-64 encoded, max length

Data type	Document that string must be
Array	One of a) a comma-separated list of values (preferred), or b) separate *name=value* parameter instances for each value of the array

API-HTTP-015

DO use the definition in the table above when translating strings.

Common Headers

Header Key	Applies to	Value
authorization	Request	{see Security Standards TBD}
accept	Request	example "application/json"
If-Match	Request	"67ab43" or * (no quotes) (see *Conditional Requests*)
If-None-Match	Request	"67ab43" or * (no quotes) (see *Conditional Requests*)
If-Modified-Since	Request	Sun, 06 Nov 1994 08:49:37 GMT (see *Conditional Requests*)
If-Unmodified-Since	Request	Sun, 06 Nov 1994 08:49:37 GMT (see *Conditional Requests*)
date	Both	Sun, 06 Nov 1994 08:49:37 GMT (see *RFC7231, Section 7.1.1.2*)
content-type	Both	example "application/merge-patch+json"
content-length	Both	1024

Header Key	Applies to	Value
ETag	Response	"67ab43" (see *Conditional Requests*)
last-modified	Response	Sun, 06 Nov 1994 08:49:37 GMT
x-wtw-error-code	Response	(see *Handling Errors*)
retry-after	Response	180 (see *RFC 7231, Section 7.1.3*)
Content-Security-Policy	Response	Should be "frame-ancestors 'none'"
Cache-Control	Response	Should be "no-cache"
Referrer-Policy	Response	Should be "no-referrer"
X-Content-Type-Options	Response	Should be "nosniff"
X-Download-Options	Response	Should be "noopen"
X-Frame-Options	Response	Should be "DENY"
X-Robots-Tag	Response	Should be "noindex, nofollow"
Content-Security-Policy	Response	Should be "default-src 'none'"

API-HTTP-016

DO support all headers shown in *italics*

API-HTTP-017

DO specify headers using kebab-casing

API-HTTP-018

DO compare request header names using case-insensitivity

API-HTTP-019

DO compare request header values using case-sensitivity if the header name requires it

API-HTTP-020

DO accept date values in headers in HTTP-Date format and return date values in headers in the IMF-fixdate format as defined in *RFC7231, Section 7.1.1.1*, e.g. "Sun, 06 Nov 1994 08:49:37 GMT".

Note: The RFC 7321 IMF-fixdate format is a "fixed-length and single-zone subset" of the RFC 1123 / RFC 5822 format, which means: a) year must be four digits, b) the seconds component of time is required, and c) the timezone must be GMT.

API-HTTP-021

DO NOT fail a request that contains an unrecognized header. Headers may be added by API gateways or middleware and this must be tolerated

API-HTTP-022

DO NOT use "x-" prefix for custom headers, unless the header already exists in production [*RFC 6648*].

Additional References - *StackOverflow - Difference between http parameters and http headers* - *Standard HTTP Headers* - *Why isn't HTTP PUT allowed to do partial updates in a REST API?*

REpresentational State Transfer (REST)

REST is an architectural style with broad reach that emphasizes scalability, generality,

independent deployment, reduced latency via caching, and security. When applying REST to your API, you define your service's resources as a collections of items. These are typically the nouns you use in the vocabulary of your service. Your service's URLs determine *the* hierarchical path developers use to perform CRUD (create, read, update, and delete) operations on resources. Note, it's important to model resource state, not behavior. There are patterns, later in these guidelines, that describe how to invoke behavior on your service. See *this article in the Azure Architecture Center* for a more detailed discussion of REST API design patterns.

When designing your service, it is important to optimize for the developer using your API.

> **API-REST-001**
>
> **DO** focus heavily on clear & consistent naming

> **API-REST-002**
>
> **DO** ensure your resource paths make sense

> **API-REST-003**
>
> **DO** simplify operations with few required query parameters & JSON fields

> **API-REST-004**
>
> **DO** establish clear contracts for string values

> **API-REST-005**
>
> **DO** use proper response codes/bodies so customer can diagnose their own problems and fix them without contacting the support or service team

Resource Schema & Field Mutability

> **API-REST-006**
>
> **DO** use the same JSON schema for PUT request/response, PATCH response, GET response, and POST request/response on a given URL path. The PATCH request schema should contain

> all the same fields with no required fields. This allows one SDK type for input/output operations and enables the response to be passed back in a request.

Field Mutability

Field Mutability	Service Request's behavior for this field
Create	Service honors field only when creating a resource. Minimize create-only fields so customers don't have to delete & re-create the resource.
Update	Service honors field when creating or updating a resource
Read	Service returns this field in a response. If the client passed a read-only field, the service **MUST** fail the request unless the passed-in value matches the resource's current value

> **API-REST-007**
>
> **DO** think about your resource's fields and how they are used.

In addition to the above, a field may be "required" or "optional". A required field is guaranteed to always exist and will typically *not* become a nullable field in a SDK's data structure. This allows customers to write code without performing a null-check. Because of this, required fields can only be introduced in the 1st version of a service; it is a breaking change to introduce required fields in a later version. In addition, it is a breaking change to remove a required field or make an optional field required or vice versa.

> **API-REST-008**
>
> **DO** make fields simple and maintain a shallow hierarchy.

> **API-REST-009**
>
> **DO** use GET for resource retrieval and return JSON in the response body

API-REST-010

YOU MAY create and update resources using PATCH [RFC5789] with JSON Merge Patch *(RFC7396)* request body.

API-REST-011

DO use PUT with JSON for wholesale create/replace operations. **NOTE:** If a v1 client PUTs a resource; any fields introduced in V2+ should be reset to their default values (the equivalent to DELETE followed by PUT).

API-REST-012

DO use DELETE to remove a resource.

API-REST-013

DO fail an operation with *400-Bad Request* if the request is improperly-formed or if any JSON field name or value is not fully understood by the specific version of the service. Return an error response as described in *Handling errors* indicating what is wrong so customer can diagnose the issue and fix it themselves.

API-REST-014

YOU MAY return secret fields via POST **if absolutely necessary**.

API-REST-015

DO NOT return secret fields via GET. For example, do not return *administratorPassword* in JSON.

API-REST-016

DO NOT add fields to the JSON if the value is easily computable from other fields to avoid bloating the body.

Create / Update / Replace Processing Rules

> **API-REST-017**
>
> **DO** follow the processing below to create/update/replace a resource:

When using this method	if this condition happens	use this response code
PATCH/PUT	Any JSON field name/value not known/valid	*400-Bad Request*
PATCH/PUT	Any Read field passed (client can't set Read fields)	*400-Bad Request*
If the resource does not exist		
PATCH/PUT	Any mandatory Create/Update field missing	*400-Bad Request*
PATCH/PUT	Create resource using Create/Update fields	*201-Created*
If the resource already exists		
PATCH	Any Create field doesn't match current value (allows retries)	*409-Conflict*
PATCH	Update resource using Update fields	*200-OK*
PUT	Any mandatory Create/Update field missing	*400-Bad Request*
PUT	Overwrite resource entirely using Create/Update fields	*200-OK*

When using this method	if this condition happens	use this response code
PATCH/PUT	The resource exists but is being regenerated / calculated	*409-Conflict*

Handling Errors

There are 2 kinds of errors:

- An error where you expect customer code to gracefully recover at runtime
- An error indicating a bug in customer code that is unlikely to be recoverable at runtime; the customer must just fix their code

API-REST-018

DO return an *x-wtw-error-code* response header with a string error code indicating what went wrong.

NOTE: x-wtw-error-code values are part of your API contract (because customer code is likely to do comparisons against them) and cannot change in the future.

API-REST-019

YOU MAY implement the *x-wtw-error-code* values as an enum with *"modelAsString": true* because it's possible add new values over time. In particular, it's only a breaking change if the same conditions result in a *different* top-level error code.

API-REST-020

YOU SHOULD NOT add new top-level error codes to an existing API without bumping the service version.

API-REST-021

DO carefully craft unique *x-wtw-error-code* string values for errors that are recoverable at

runtime. Reuse common error codes for usage errors that are not recoverable.

API-REST-022

YOU MAY group common customer code errors into a few *x-wtw-error-code* string values.

API-REST-023

DO ensure that the top-level error's *code* value is identical to the *x-wtw-error-code* header's value.

API-REST-024

DO provide a response body with the following structure:

ErrorResponse : Object

Property	Type	Required	Description
error	ErrorDetail	✔	The top-level error object whose *code* matches the *x-wtw-error-code* response header

ErrorDetail : Object

Property	Type	Required	Description
code	String	✔	One of a server-defined set of error codes.
message	String	✔	A human-readable representation of the error.
target	String		The target of the error.
details	ErrorDetail[]		An array of details about specific errors that led to this reported error.

Property	Type	Required	Description
innererror	InnerError		An object containing more specific information than the current object about the error.
additional properties			Additional properties that can be useful when debugging.

InnerError : Object

Property	Type	Required	Description
code	String		A more specific error code than was provided by the containing error.
innererror	InnerError		An object containing more specific information than the current object about the error.

Example:

```
{
  "error" :  {
    "code": "InvalidPasswordFormat",
    "message": "Human-readable description",
    "target": "target of error",
    "innererror": {
      "code": "PasswordTooShort",
      "minLength": 6,
    }
  }
}
```

API-REST-025

DO document the service's top-level error code strings; they are part of the API contract.

API-REST-026

YOU MAY treat the other fields as you wish as they are *not* considered part of your service's API contract and customers should not take a dependency on them or their value. They exist to help customers self-diagnose issues.

API-REST-027

YOU MAY add additional properties for any data values in your error message so customers don't resort to parsing your error message. For example, an error with *"message": "A maximum of 16 keys are allowed per account."* might also add a *"maximumKeys": 16* property. This is not part of your API contract and should only be used for diagnosing problems.

Note: Do not use this mechanism to provide information developers need to rely on in code (ex: the error message can give details about why you've been throttled, but the Retry-After should be what developers rely on to back off).

API-REST-028

YOU SHOULD NOT document specific error status codes in your OpenAPI/Swagger spec unless the "default" response cannot properly describe the specific error response (e.g. body schema is different).

API-REST-029

DO NOT disclose any information about internal errors. Internal errors should be logged but only an error code should be returned to the consumer.

Note: you can only control the errors return from components within the developers control this includes the application code and API Gateway, however components like CDNs or WAFs will return system specific error codes to the consumer.

JSON

API-JSON-001

DO use camel case for all JSON field names. Do not upper-case acronyms; use camel case.

API-JSON-002

DO treat JSON field names with case-sensitivity.

API-JSON-003

DO treat JSON field values with case-sensitivity. There may be some exceptions (e.g. GUIDs) but avoid if at all possible.

Services, and the clients that access them, may be written in multiple languages. To ensure interoperability, JSON establishes the "lowest common denominator" type system, which is always sent over the wire as UTF-8 bytes. This system is very simple and consists of three types:

Type	Description
Boolean	true/false (always lowercase)
Number	Signed floating point (IEEE-754 binary64; int range: -253+1 to +253-1)
String	Used for everything else

API-JSON-004

DO use integers within the acceptable range of JSON number.

API-JSON-005

DO establish a well-defined contract for the format of strings. For example, determine maximum length, legal characters, case-(in)sensitive comparisons, etc. Where possible, use standard formats, e.g. RFC3339 for date/time.

API-JSON-006

DO use strings formats that are well-known and easily parsable/formattable by many

programming languages, e.g. RFC3339 for date/time.

API-JSON-007

DO ensure that information exchanged between your service and any client is "round-trippable" across multiple programming languages.

API-JSON-008

DO use *RFC3339* for date/time.

API-JSON-008

DO use a fixed time interval to express durations e.g., milliseconds, seconds, minutes, days, etc., and include the time unit in the property name e.g., *backupTimeInMinutes* or *ttlSeconds*.

API-JSON-010

YOU MAY use *RFC3339 time intervals* only when users must be able to specify a time interval that may change from month to month or year to year e.g., "P3M" represents 3 months no matter how many days between the start and end dates, or "P1Y" represents 366 days on a leap year. The value must be round-trippable.

API-JSON-011

DO use *RFC4122* for UUIDs.

API-JSON-012

YOU MAY use JSON objects to group sub-fields together.

API-JSON-013

YOU MAY use JSON arrays if maintaining an order of values is required. Avoid arrays in other situations since arrays can be difficult and inefficient to work with, especially with JSON Merge Patch where the entire array needs to be read prior to any operation being

applied to it.

API-JSON-014

YOU SHOULD use JSON objects instead of arrays whenever possible.

Enums & SDKs (Client libraries)

It is common for strings to have an explicit set of values. These are often reflected in the OpenAPI definition as enumerations. These are extremely useful for developer tooling, e.g. code completion, and client library generation. :::

However, it is not uncommon for the set of values to grow over the life of a service. For this reason, Microsoft's tooling uses the concept of an "extensible enum," which indicates that the set of values should be treated as only a *partial* list. This indicates to client libraries and customers that values of the enumeration field should be effectively treated as strings and that undocumented value may returned in the future. This enables the set of values to grow over time while ensuring stability in client libraries and customer code.

API-JSON-015

YOU SHOULD use extensible enumerations unless you are positive that the symbol set will NEVER change over time.

API-JSON-016

DO document to customers that new values may appear in the future so that customers write their code today expecting these new values tomorrow.

API-JSON-017

DO NOT remove values from your enumeration list as this breaks customer code.

Polymorphic types

API-JSON-18

> **YOU SHOULD NOT** use polymorphic JSON types because they greatly complicate the customer code due to runtime dynamic casts and the introduction of new types in the future.

If you can't avoid them, then follow the guideline below.

> **API-JSON-019**
>
> **DO** define a *kind* field indicating the kind of the resource and include any kind-specific fields in the body.

Below is an example of JSON for a Rectangle and Circle:

Rectangle

```
{
   "kind": "rectangle",
   "x": 100,
   "y": 50,
   "width": 10,
   "length": 24,
   "fillColor": "Red",
   "lineColor": "White",
   "subscription": {
      "kind": "free"
   }
}
```

Circle

```
{
   "kind": "circle",
   "x": 100,
   "y": 50,
   "radius": 10,
   "fillColor": "Green",
   "lineColor": "Black",
   "subscription": {
      "kind": "paid",
```

```
      "expiration": "10",
      "invoice": "1",
   }
}
```

Both Rectangle and Circle have common fields: *kind*, *fillColor*, *lineColor*, and *subscription*. A Rectangle also has *x*, *y*, *width*, and *length* while a Circle has *x*, *y*, and *radius*. The *subscription* is a nested polymorphic type. A *free* subscription has no additional fields and a *paid* subscription has *expiration* and *invoice* fields.

Common API Patterns

Collections

API-COLLECTIONS-001

DO structure the response to a list operation as an object with a top-level array field containing the set (or subset) of resources.

API-COLLECTIONS-002

YOU SHOULD support paging today if there is ever a chance in the future that the number of items can grow to be very large.

NOTE: It is a breaking change to add paging in the future

API-COLLECTIONS-003

YOU MAY expose an operation that lists your resources by supporting a GET method with a URL to a resource-collection (as opposed to a resource-id).

Example Response Body

```
{
    "value": [
       { "id": "Item 01", "etag": "0xabc", "price": 99.95, "sizes": null },
```

```
        { … },
        { … },
        { "id": "Item 99", "etag": "0xdef", "price": 59.99, "sizes": null }
    ],
    "nextLink": "{opaqueUrl}"
}
```

API-COLLECTIONS-004

DO include the *id* field and *etag* field (if supported) for each item as this allows the customer to modify the item in a future operation.

API-COLLECTIONS-005

DO clearly document that resources may be skipped or duplicated across pages of a paginated collection unless the operation has made special provisions to prevent this (like taking a time-expiring snapshot of the collection).

API-COLLECTIONS-006

DO return a *nextLink* field with an absolute URL that the client can GET in order to retrieve the next page of the collection.

Note: The service is responsible for performing any URL-encoding required on the *nextLink* URL.

API-COLLECTIONS-007

DO include any query parameters required by the service in *nextLink*.

API-COLLECTIONS-008

YOU SHOULD use *value* as the name of the top-level array field unless a more appropriate name is available.

API-COLLECTIONS-009

DO NOT return the *nextLink* field at all when returning the last page of the collection.

API-COLLECTIONS-010

DO NOT return the *nextLink* field with a value of null.

API-COLLECTIONS-011

YOU SHOULD NOT return a *count* of all objects in the collection as this may be expensive to compute.

Query options

Parameter name	Type	Description
filter	string	an expression on the resource type that selects the resources to be returned
orderby	string array	a list of expressions that specify the order of the returned resources
skip	integer	an offset into the collection of the first resource to be returned
top	integer	the maximum number of resources to return from the collection
maxpagesize	integer	the maximum number of resources to include in a single response
select	string array	a list of field names to be returned for each resource
expand	string array	a list of the related resources to be included in line with each resource

API-COLLECTIONS-012

YOU MAY support the defined (above) query parameters allowing customers to control the list operation.

API-COLLECTIONS-013

DO return an error if the client specifies any parameter not supported by the service.

API-COLLECTIONS-014

DO treat these query parameter names as case-sensitive.

API-COLLECTIONS-015

DO apply *select* or *expand* options after applying all the query options in the table above.

API-COLLECTIONS-016

DO apply the query options to the collection in the order shown in the table above.

API-COLLECTIONS-017

DO NOT prefix any of these query parameter names with "$" (the convention in the *OData standard*).

filter

API-COLLECTIONS-018

YOU MAY support filtering of the results of a list operation with the *filter* query parameter.

The value of the *filter* option is an expression involving the fields of the resource that produces a Boolean value. This expression is evaluated for each resource in the collection and only items where the expression evaluates to true are included in the response.

> **API-COLLECTIONS-019**
>
> **DO** omit all resources from the collection for which the *filter* expression evaluates to false or to null, or references properties that are unavailable due to permissions.

Example: return all Products whose Price is less than $10.00

```
GET https://api.contoso.com/products?`filter`=price lt 10.00
```

***filter* operators**

Operator	Description	Example
Comparison Operators		
eq	Equal	city eq 'Redmond'
ne	Not equal	city ne 'London'
gt	Greater than	price gt 20
ge	Greater than or equal	price ge 10
lt	Less than	price lt 20
le	Less than or equal	price le 100
Logical Operators		
and	Logical and	price le 200 and price gt 3.5
or	Logical or	price le 3.5 or price gt 200

Operator	Description	Example
not	Logical negation	not price le 3.5
Grouping Operators		
( )	Precedence grouping	(priority eq 1 or city eq 'Redmond') and price gt 100

API-COLLECTIONS-020

YOU MAY support ICT specified operators (see table) in filter expressions

API-COLLECTIONS-021

DO respond with an error message as defined in the *Handling Errors* section if a client includes an operator in a *filter* expression that is not supported by the operation.

Operator Precedence

Group	Operator	Description
Grouping	( )	Precedence grouping
Unary	not	Logical Negation
Relational	gt	Greater Than
	ge	Greater than or Equal
	lt	Less Than
	le	Less than or Equal
Equality	eq	Equal

Group	Operator	Description
	ne	Not Equal
Conditional AND	and	Logical And
Conditional OR	or	Logical Or

API-COLLECTIONS-022

DO use the ICT specified operator precedence(see table) for supported operators when evaluating *filter* expressions. Operators are listed by category in order of precedence from highest to lowest. Operators in the same category have equal precedence and should be evaluated left to right:

API-COLLECTIONS-023

YOU MAY support orderby and *filter* functions such as concat and contains. For more information, see *odata Canonical Functions*.

Operator examples

The following examples illustrate the use and semantics of each of the logical operators. :::

Example: all products with a name equal to 'Milk'

GET https://api.contoso.com/products?`filter`=name eq 'Milk'

Example: all products with a name not equal to 'Milk'

GET https://api.contoso.com/products?`filter`=name ne 'Milk'

Example: all products with the name 'Milk' that also have a price less than 2.55:

GET https://api.contoso.com/products?`filter`=name eq 'Milk' and price lt 2.55

Example: all products that either have the name 'Milk' or have a price less than 2.55:

GET https://api.contoso.com/products?`filter`=name eq 'Milk' or price lt 2.55

Example: all products that have the name 'Milk' or 'Eggs' and have a price less than 2.55:

GET https://api.contoso.com/products?`filter`=(name eq 'Milk' or name eq 'Eggs') and price lt 2.55

orderby

API-COLLECTIONS-024

YOU MAY support sorting of the results of a list operation with the *orderby* query parameter.

NOTE: It is unusual for a service to support orderby because it is very expensive to implement as it requires sorting the entire large collection before being able to return any results.

The value of the *orderby* parameter is a comma-separated list of expressions used to sort the items. A special case of such an expression is a property path terminating on a primitive property.

Each expression in the *orderby* parameter value may include the suffix "asc" for ascending or "desc" for descending, separated from the expression by one or more spaces.

API-COLLECTIONS-025

DO sort the collection in ascending order on an expression if "asc" or "desc" is not specified.

API-COLLECTIONS-026

DO sort NULL values as "less than" non-NULL values.

API-COLLECTIONS-027

DO sort items by the result values of the first expression, and then sort items with the same value for the first expression by the result value of the second expression, and so on.

API-COLLECTIONS-028

DO use the inherent sort order for the type of the field. For example, date-time values

should be sorted chronologically and not alphabetically.

API-COLLECTIONS-029

DO respond with an error message as defined in the *Handling Errors* section if the client requests sorting by a field that is not supported by the operation.

For example, to return all people sorted by name in ascending order:

GET https://api.contoso.com/people?orderby=name

For example, to return all people sorted by name in descending order and a secondary sort order of hireDate in ascending order.

GET https://api.contoso.com/people?orderby=name desc,hireDate

Sorting MUST compose with _filter_ing such that:

GET https://api.contoso.com/people?`filter`=name eq 'david'&orderby=hireDate

will return all people whose name is David sorted in ascending order by hireDate.

Considerations for sorting with pagination

API-COLLECTIONS-030

DO use the same _filter_ing options and sort order for all pages of a paginated list operation response.

skip

API-COLLECTIONS-031

DO define the *skip* parameter as an integer with a default and minimum value of 0.

API-COLLECTIONS-032

YOU MAY allow clients to pass the *skip* query parameter to specify an offset into collection

of the first resource to be returned.

top

API-COLLECTIONS-033

YOU MAY allow clients to pass the *top* query parameter to specify the maximum number of resources to return from the collection.

If supporting *top*:

API-COLLECTIONS-034

DO define the *top* parameter as an integer with a minimum value of 1. If not specified, *top* has a default value of infinity.

API-COLLECTIONS-035

DO return the collection's *top* number of resources (if available), starting from *skip*.

maxpagesize

API-COLLECTIONS-036

YOU MAY allow clients to pass the *maxpagesize* query parameter to specify the maximum number of resources to include in a single page response.

API-COLLECTIONS-037

DO define the *maxpagesize* parameter as an optional integer with a default value appropriate for the collection.

API-COLLECTIONS-038

DO make clear in documentation of the *maxpagesize* parameter that the operation may choose to return fewer resources than the value specified.

API Versioning

ICT services need to change over time. And to achieve this the API Contract should be versioned using sematic versioning. When changing a service, there are 2 requirements:

1. Already-running customer workloads must not break due to a service change
2. With the exception of a **major version change**, Customers can adopt a new service version without requiring any code changes (Of course, the customer must modify code to leverage any new service features.)

This means that for changes to the API interface that are classified as minor, the existing API contract can be entirely replaced with the newer contract as the contract maintains backward compatibility. For major version changes, both instance of the contract must be made available if any consumers are still using the older version. Ways to implement this in your service are discussed in Consideration for Service Design

API-VERSIONING-001

DO use *sematic versioning* for APIs

API-VERSIONING-002

DO include only **major version** of the **API** in the URL as part of the base path. The version is for the API and not the individual resource.

Example: for the AMM model-management API version V1.2.8

```
PUT https://mycompany.ictplatform.co.uk/amm/model-management/v1/models
/13123
```

See Uniform Resource Locators for more information

API-VERSIONING-003

DO create a new Open API Specification document for **any change to the API contract**, this document should be versioned in a version control system with the full semantic version as a tag.

Use Extensible Enums

While removing a value from an enum is a breaking change, adding value to an enum can be handled with an *extensible enum*. An extensible enum is a string value that has been marked with a special marker - setting *modelAsString* to true within an *x-ms-enum* block. For example:

```
"createdByType": {
   "type": "string",
   "description": "The type of identity that created the resource.",
   "enum": [
      "User",
      "Application",
      "ManagedIdentity",
      "Key"
   ],
   "x-ms-enum": {
      "name": "createdByType",
      "modelAsString": true
   }
}
```

API-VERSIONING-004

YOU SHOULD use extensible enums unless you are positive that the symbol set will **NEVER** change over time.

API-VERSIONING-005

DO use the next full semantic version number with `-preview<n>` suffix for preview versions for the API.

An example of a preview version would be `1.0.0-preview1` for the first preview, `1.0.0-`

`preview2` for the second preview of API version `1.0.0` .

Deprecating Behavior Notification

When the *API Versioning* guidance above results in a new major version of a specific API and this is approved by the *API Stewardship board* this must be communicated to its callers.

All efforts should be made to deprecate old major version of an API. To determine whether the API version should continue to be supported or be deprecated please discuss with the Product Owner and the API Stewardship board. If the decision is made to deprecate the API version then the *wtw-deprecating* response header must be added to the response with a semicolon-delimited string notifying the caller what is being deprecated, when it will no longer function, and a URL linking to more information such as what new operation they should use instead.

The purpose is to inform customers (when debugging/logging responses) that they must take action to modify their call to the service's operation and use a newer API version or their call will soon stop working entirely. It is not expected that client code will examine/parse this header's value in any way; it is purely informational to a human being. The string is *not* part of an API contract (except for the semi-colon delimiters) and may be changed/improved at any time without incurring a breaking change.

> **API-PATTERNS-001**
>
> **DO** include the *wtw-deprecating* header in the operation's response *only if* the operation will stop working in the future and the client *must take* action in order for it to keep working. > NOTE: We do not want to scare customers with this header.

> **API-PATTERNS-002**
>
> **DO** make the header's value a semicolon-delimited string indicating a set of deprecations where each one indicates what is deprecating, when it is deprecating, and a URL to more information.

Deprecations should use the following pattern:

<description> will retire on <date> (`url`);

Where the following placeholders should be provided: - *description*: a human-readable description of what is being deprecated - *date*: the target date that this will be deprecated. This should be expressed following the format in *ISO 8601*, e.g. "2022-10-31". - *url*: a fully qualified url that the user can follow to learn more about what is being deprecated.

For example:

wtw-deprecating: API version 1 will retire on 2022-12-01 (https://ictplatform.co.uk/client-portal/notification-1);TLS 1.0 & 1.1 will retire on 2020-10-30 (https://ictplatform.co.uk/client-portal/notification-2)

> **API-PATTERNS-003**
>
> **DO NOT** introduce *wtw-deprecating header without approval from the Product Owner and*API Stewardship board_.

Repeatability of requests

The ability to retry failed requests for which a client never received a response greatly simplifies the ability to write resilient distributed applications. While HTTP designates some methods as safe and/or idempotent (and thus retryable), being able to retry other operations such as create-using-POST-to-collection is desirable.

API-PATTERNS-004

YOU SHOULD support repeatable requests according as defined in *OASIS Repeatable Requests Version 1.0*.

- The tracked time window (difference between the *Repeatability-First-Sent* value and the current time) **MUST** be at least 5 minutes.
- A service advertises support for repeatability requests by adding the *Repeatability-First-Sent* and *Repeatability-Request-ID* to the set of headers for a given operation.
- When understood, all endpoints co-located behind a DNS name **MUST** understand the header. This means that a service **MUST NOT** ignore the presence of a header for any endpoints behind the DNS name, but rather fail the request containing a *Repeatability-Request-ID* header if that particular endpoint lacks support for repeatable requests. Such partial support **SHOULD** be avoided due to the confusion it causes for clients.

Long-Running Operations & Jobs

When the processing for an operation may take a significant amount of time to complete, it should be implemented as a *long-running operation (LRO)*. This allows clients to continue running while the operation is being processed. The client obtains the outcome of the operation at some later time through another API call. See the *Considerations for Service Design* in Considerations for Service Design for an introduction to the design of long-running operations.

API-PATTERNS-005

DO implement an operation as an LRO if the 99th percentile response time is greater than 1s.

API-PATTERNS-006

DO NOT implement PATCH as an LRO. If LRO update is required it must be implemented with POST.

In rare instances where an operation may take a *very long* time to complete, e.g. longer than 15 minutes, it may be better to expose this as a first class resource of the API rather than as an operation on another resource.

There are two basic patterns suggested for long-running, API fronted operations. The first pattern is used for a POST and DELETE operations that initiate the LRO. These return a *202 Accepted* response with a JSON status monitor in the response body. The second pattern applies only in the case of a PUT operation to create a resource that also involves additional long-running processing. For guidance on when to use a specific pattern, please refer to *Considerations for Service Design*. These are described in the following two sections.

POST or DELETE LRO pattern

A POST or DELETE long-running operation accepts a request from the client to initiate the operation processing and returns a *status monitor* that reports the operation's progress.

> **API-PATTERNS-007**
>
> **DO NOT** use a long-running POST to create a resource – use PUT as described below.

> **API-PATTERNS-008**
>
> **DO** allow the client to pass an *Operation-Id* header with an ID for the operation's status monitor.

> **API-PATTERNS-009**
>
> **DO** generate an ID (typically a GUID) for the status monitor if the *Operation-Id* header was not passed by the client.

> **API-PATTERNS-010**
>
> **DO** fail a request with a *400-BadRequest* if the *Operation-Id* header matches an existing operation unless the request is identical to the prior request (a retry scenario).

> **API-PATTERNS-011**
>
> **DO** perform as much validation as practical when initiating the operation to alert clients of errors early.

API-PATTERNS-012

DO return a *202-Accepted* status code from the request that initiates an LRO if the processing of the operation was successfully initiated (except for "PUT with additional processing" type LRO).

API-PATTERNS-013

YOU SHOULD NOT return any other *2xx* status code from the initial request of an LRO – return *202-Accepted* and a status monitor even if processing was completed before the initiating request returns.

API-PATTERNS-014

DO return a status monitor in the response body as described in *Obtaining status and results of long-running operations*.

API-PATTERNS-015

YOU SHOULD include an *Operation-Location* header in the response with the absolute URL of the status monitor for the operation.

PUT operation with additional long-running processing

For a PUT (create or replace) with additional long-running processing:

API-PATTERNS-016

DO allow the client to pass an *Operation-Id* header with a ID for the status monitor for the operation.

API-PATTERNS-017

DO generate an ID (typically a GUID) for the status monitor if the *Operation-Id* header was not passed by the client.

API-PATTERNS-018

DO fail a request with a *400-BadRequest* if the *Operation-Id* header that matches an existing operation unless the request is identical to the prior request (a retry scenario).

API-PATTERNS-019

DO perform as much validation as practical when initiating the operation to alert clients of errors early.

API-PATTERNS-020

DO return a *201-Created* status code for create or *200-OK* for replace from the initial request with a representation of the resource if the resource was created successfully.

API-PATTERNS-021

DO include an *Operation-Id* header in the response with the ID of the status monitor for the operation.

API-PATTERNS-022

DO include response headers with any additional values needed for a GET request to the status monitor (e.g. location).

API-PATTERNS-023

YOU SHOULD include an *Operation-Location* header in the response with the absolute URL of the status monitor for the operation.

Obtaining status and results of long-running operations

For all long-running operations, the client will issue a GET on a status monitor resource to obtain the current status of the operation.

API-PATTERNS-024

DO support the GET method on the status monitor endpoint that returns a *200-OK* response with the current state of the status monitor.

API-PATTERNS-025

DO return a status monitor in the response body that conforms with the structure below.

OperationStatus : Object

Property	Type	Required	Description
id	string	true	The unique id of the operation
status	string	true	enum that includes values "NotStarted", "Running", "Succeeded", "Failed", and "Canceled"
error	ErrorDetail		Error object that describes the error when status is "Failed"
result	object		Only for POST action-type LRO, the results of the operation when completed successfully
additionalproperties			Additional named or dynamic properties of the operation

API-PATTERNS-026

DO include the *id* of the operation and any other values needed for the client to form a GET request to the status monitor (e.g. a *location* path parameter).

API-PATTERNS-027

DO include a *Retry-After* header in the response to GET requests to the status monitor if the operation is not complete. The value of this header should be an integer number of seconds

to wait before making the next request to the status monitor.

API-PATTERNS-028

DO include the *result* property (if any) in the status monitor for a POST action-type long-running operation when the operation completes successfully.

API-PATTERNS-029

DO NOT include a *result* property in the status monitor for a long-running operation that is not a POST action-type long-running operation.

API-PATTERNS-030

DO retain the status monitor resource for some publicly documented period of time (at least 24 hours) after the operation completes.

Conditional Requests

When designing an API, you will almost certainly have to manage how your resource is updated. For example, if your resource is a bank account, you will want to ensure that one transaction–say depositing money–does not overwrite a previous transaction. Similarly, it could be very expensive to send a resource to a client. This could be because of its size, network conditions, or a myriad of other reasons. To enable this level of control, services should leverage an *ETag* header, or "entity tag," which will identify the 'version' or 'instance' of the resource a particular client is working with. An *ETag* is always set by the service and will enable you to *conditionally* control how your service responds to requests, enabling you to provide predictable updates and more efficient access.

API-PATTERNS-031

YOU SHOULD return an *ETag* with any operation returning the resource or part of a resource or any update of the resource (whether the resource is returned or not).

API-PATTERNS-032

> **YOU SHOULD** use *ETag_s consistently across your API, i.e. if you use an_ETag*, accept it on all other operations.

You can learn more about conditional requests by reading *RFC7232*.

Cache Control

One of the more common uses for *ETag* headers is cache control, also referred to a "conditional GET." This is especially useful when resources are large in size, expensive to compute/calculate, or hard to reach (significant network latency). That is, using the value of the *ETag* , the server can determine if the resource has changed. If there are no changes, then there is no need to return the resource, as the client already has the most recent version.

Implementing this strategy is relatively straightforward. First, you will return an *ETag* with a value that uniquely identifies the instance (or version) of the resource. The *Computing ETags* section provides guidance on how to properly calculate the value of your *ETag*. In these scenarios, when a request is made by the client an *ETag* header is returned, with a value that uniquely identifies that specific instance (or version) of the resource. The *ETag* value can then be sent in subsequent requests as part of the *If-None-Match* header. This tells the service to compare the *ETag* that came in with the request, with the latest value that it has calculated. If the two values are the same, then it is not necessary to return the resource to the client–it already has it. If they are different, then the service will return the latest version of the resource, along with the updated *ETag* value in the header.

> ⓘ **API-PATTERNS-033**
>
> **YOU SHOULD** implement conditional read strategies

When supporting conditional read strategies:

GET Request	Return code	Response
ETag value = *If-None-Match* value	*304-Not Modified*	no additional information

GET Request	Return code	Response
ETag value != *If-None-Match* value	*200-OK*	Response body include the serialized value of the resource (typically JSON)

API-PATTERNS-034

DO adhere to the guidance (above) when supporting conditional read strategies

For more control over caching, please refer to the *cache-control__HTTP header*.

Optimistic Concurrency

An *ETag* should also be used to reflect the create, update, and delete policies of your service. Specifically, you should avoid a "pessimistic" strategy where the 'last write always wins." These can be expensive to build and scale because avoiding the "lost update" problem often requires sophisticated concurrency controls. Instead, implement an "optimistic concurrency" strategy, where the incoming state of the resource is first compared against what currently resides in the service. Optimistic concurrency strategies are implemented through the combination of *ETags* and the *HTTP Request / Response Pattern*.

API-PATTERNS-035

YOU SHOULD NOT implement pessimistic update strategies, e.g. last writer wins.

When supporting optimistic concurrency:

Operation	Header	Value	ETag check	Return code	Response
PATCH / PUT	*If-None-Match*	*	check for *any* version of the resource ('*' is a wildcard used to match anything), if none are found, create the	*200-OK* or *201-Created*	Response header MUST include the new *ETag* value. Response body SHOULD include the serialized value

Operation	Header	Value	ETag check	Return code	Response
			resource.		of the resource (typically JSON).
PATCH / PUT	*If-None-Match*	*	check for *any* version of the resource, if one is found, fail the operation	*412-Precondition Failed*	Response body SHOULD return the serialized value of the resource (typically JSON) that was passed along with the request.
PATCH / PUT	*If-Match*	value of ETag	value of *If-Match* equals the latest ETag value on the server, confirming that the version of the resource is the most current	*200-OK* or *201-Created*	Response header MUST include the new *ETag* value. Response body SHOULD include the serialized value of the resource (typically JSON).
PATCH / PUT	*If-Match*	value of ETag	value of *If-Match* header DOES NOT equal the latest ETag value on the server, indicating a change has ocurred since after the client fetched the resource	*412-Precondition Failed*	Response body SHOULD return the serialized value of the resource (typically JSON) that was passed along with the request.
DELETE	*If-*	value	value matches the	*204-No*	Response body

Operation	Header	Value	ETag check	Return code	Response
	Match	of ETag	latest value on the server	*Content*	SHOULD be empty.
DELETE	*If-Match*	value of ETag	value does NOT match the latest value on the server	*412-Preconditioned Failed*	Response body SHOULD be empty.

API-PATTERNS-036

DO adhere to the ICT guidance (see table) when supporting optimistic concurrency.

Computing ETags

The strategy that you use to compute the *ETag* depends on its semantic. For example, it is natural, for resources that are inherently versioned, to use the version as the value of the *ETag*. Another common strategy for determining the value of an *ETag* is to use a hash of the resource. If a resource is not versioned, and unless computing a hash is prohibitively expensive, this is the preferred mechanism.

API-PATTERNS-037

YOU SHOULD use a hash of the representation of a resource rather than a last modified/version number

While it may be tempting to use a revision/version number for the resource as the ETag, it interferes with client's ability to retry update requests. If a client sends a conditional update request, the service acts on the request, but the client never receives a response, a subsequent identical update will be seen as a conflict even though the retried request is attempting to make the same update.

API-PATTERNS-038

YOU SHOULD , if using a hash strategy, hash the entire resource.

API-PATTERNS-039

YOU SHOULD , if supporting range requests, use a strong ETag in order to support caching.

API-PATTERNS-040

YOU MAY use or, include, a timestamp in your resource schema. If you do this, the timestamp shouldn't be returned with more than subsecond precision, and it SHOULD be consistent with the data and format returned, e.g. consistent on milliseconds.

API-PATTERNS-041

YOU MAY consider Weak ETags if you have a valid scenario for distinguishing between meaningful and cosmetic changes or if it is too expensive to compute a hash.

API-PATTERNS-042

DO , when supporting multiple representations (e.g. Content-Encodings) for the same resource, generate different ETag values for the different representations.

Distributed Tracing & Telemetry

The ICT Observability standards specify that services must send telemetry and distributed tracing information on every request. Telemetry information is vital to the effective operation of your service and should be a consideration from the outset of design and implementation efforts.

API-PATTERNS-043

DO follow the the ICT Observability standards for supporting telemetry headers.

API-PATTERNS-044

DO NOT reject a call if you have custom headers you don't understand, and specifically, distributed tracing headers.

Secure Query Parameters

Securing

Final thoughts

These guidelines describe the upfront design considerations, technology building blocks, and common patterns that ICT teams encounter when building an API for their service. There is a great deal of information in them that can be difficult to follow. The ICT API Stewardship board will provide support to ensuring your success.

The ICT API Stewardship board is a collection of dedicated architects that are focused on helping ICT service teams build interfaces that are intuitive, maintainable, consistent, and most importantly, great for our customers to use. Because APIs affect nearly all downstream decisions, you are encouraged to reach out to the Stewardship board early in the development process. These architects will work with you to apply these guidelines and identify any hidden pitfalls in your design. For more information on how to part with the Stewardship board, please refer to *Considerations for Service Design*.

Edit this page

*Last updated on **May 23, 2023** by **Adam Davis***