Was wondering if a principle for Versioning and a principle for Sorting are needed?

1. Scope

This OGC document presents a set of common design principles for developing OGC Web API standards

The use of agreed to common elements and principals for OGC Web APIs allows for a common pattern to ensure consistency of OGC Web APIs, enhanced interoperability, and easier adoption.

There are common design principles in main-stream IT that should be adopted to ease the development, adoption, and use of OGC Web APIs.

Even so there are some aspects related to OGC Web API specification that need to be agreed upon to ensure consistent APIs for different geospatial technology, information domains, and thematic topics currently used in the OGC.

In particular, the goal is to avoid as much as possible OGCAPIs that are fundamentally different for accessing, processing and managing different geospatial resources such as Features, Maps, Tiles, Coverages, Observations, Processes, etc.

2. Conformance

This OGC Architecture Board Position Document defines a series of requirements for developing and documenting OGC API standards.

Conformance with this policy shall be checked manually through review or through automation where possible.

A checklist is provided to each active Standards Working Group (SWG) to document how any OGC Web API being developed or revised complies with the guidelines and if any exemptions to the guidelines apply.

Completed checklists are to be reviewed by the OAB as part of the process for a candidate OGC Web API specification to be considered for Public Request for Comment (RFC).

6. Guidelines

6.1. Purpose and Process

The implementation of Web APIs that allow the management, processing and use of geospatial information should be possible by anyone familiar with a Web APIs designed for main stream IT. However, when an OGC Web API is being designed by multiple domain experts, while trying to address multipurpose usage, ensuring that a common design pattern is used among and between all SWGs can be challenging..

To ensure that (i) OGC Web API design and specification across all different domains of expertise is coherent and (ii) the maximum main stream IT design patterns are reused, the OGC Architecture Board (OAB) requested elaboration and documentation of OGC Web API development guidelines. This process was performed in the Architecture DWG with the collaboration of the OWS Common SWG. At this time, this is a living document formulated to inspire additional discussion and refinement within and between OGC Working Groups and to contribute our learnings and suggestions to the broader technology community that depends on geospatial resources. The final aim is to reach consensus and converge in a document that the OGC Members can approve as a Policy Document.

Even though the main goal is to provide OGC Web API development guidance, a key secondary objective is to have a checklist to streamline the OAB and Technical Committee review process. The assessment for verifying the Web API design against the checklist should be submitted with the Web API draft standard to the OAB. While following all principles is not mandatory, reasons for deviation should be given.

For the moment please consider document to be the foundation for further discussion and consensus. Please create or comment on existing Issues to discuss changes, corrections, and enhancements to the principles.

6.2. Starting point

The starting point for developing the Design Principles listed is this document was a presentation on OGC Web API Design Principles [requires OGC portal login](https://portal.opengeospatial.org/files/?artifact_id=78344) given during the OGC TC meetings in Orleans, France and Fort Collins, Colorado USA. The presentation summarized a collection of the Web API design principles used by major players in main stream IT business. The purpose of the presentation was to ensure that the "common part of an API" is designed such that it can be re-used and adopted easily. However, the initial presentation was incomplete and there is room for a good consensus discussion.

The original author of the presentation (Andreas Matheus), in collaboration with Charles Heazel, agreed to make the content available in this open GitHub repo for the purpose of creating a starting point in discussion and deriving a set of guidelines that could eventually be used to test OGC Web API Implementation Standards for conformance.

6.3. Design Principles

6.3.1. Principle #1 – Don’t Reinvent

For aspects and functional capabilities that are already solved in main-stream IT and meet geospatial community requirements, simply adopt these API elements.

Focus instead on geo-centric and domain specific requirements to create new APIs or extend existing APIs.

6.3.2. Principle #2 – Keep the API Simple and Intuitive

Make the developer of the API successful as quickly as possible!

6.3.3. Principle #3 - Use Well-Known Resource Types

Identify resource types and reuse existing definitions from the OGC Naming Authority resource type register (to be established).

Encodings of resource types should be associated with an IANA registered media type. See Principal 16.

6.3.4. Principle #4 – Construct consistent URIs

Great Web APIs look like they were designed by a single team. The most obvious properties of an API are the access paths and the URL templates which define them. Therefore, OGC conventions for the construction of access path templates are essential. Some of these templates are emerging though the Web Feature Service 3.0 efforts. Before creating a new URI scheme, the SWG should follow and build on existing OGC approaches that have proven to work. If the SWG us creating a URI scheme, please explain the URI pattern.

The API URI pattern should be documented, formalized, explained.

One existing approach in the OGC is the following (simplified):

For resource types that consist of a collection of resources, the pattern at the end of the URI path is as follows where resourceType is in plural:

.../{resourceType}/{resourceId}

Where resources are nested, the path elements may be concatenated. For example:

.../collections - returns the list of feature collections

.../collections/highways - returns representation of the collection 'highways'

.../collections/highways/items - returns the features in the collection 'highways'

.../collections/highways/items/A8 - returns the feature 'A9' in the collection 'highways'

For resource types that consist of a single resource, the pattern at the end of the URI path is as follows where resourceType is in singular:

.../{resourceType}

For example:

.../collections/highways/schema - returns the schema for the features in the collection 'highways'

.../collections/highways/metadata - returns the information about the features in the collection 'highways'

Note that it does not matter if singular or plural is used for nouns to build the paths, but use a consistent pattern throughout the API!

6.3.5. Principle #5 – Use HTTP Methods consistent with RFC 2616

Include in the API design the use of all HTTP methods that operate on resources: **GET, POST, PUT, DELETE.**

Define the semantics carefully when a method is invoked on a particular URI addressing a resource. For example:

| **Resource** | **POST** | **GET** | **PUT** | **DELETE** |
| --- | --- | --- | --- | --- |
| ../collections/highways/items | create a new highway | list all highways | bulk update of highways | delete all highways |
| …​/collections/highways/items/A8 | Error! | show A8 | If exists: Update A8 else: Create A8 | delete highway A8 |

Do not force all semantics in just HTTP GET!

Also consider support for other HTTP methods:

* HEAD to return HTTP Headers with no payload;
* OPTIONS to support W3C CORS;
* PATCH to update parts of an existing resource.

6.3.6. Principle #6 – Put Selection Criteria behind the ‘?’

The radical idea behind the '?' concept is that everything **left** of the **'?'** (the path design) identifies a resource and that everything **right** of the **'?'** may select specific representation(s) of parts or the entire resource.

For example:

.../collections/highways/items?id=A8 => returns highway A8

.../collections/highways/items?id=A8,A9 => returns highways A8 and A9

.../collections/highways/items/A8?time=2019-02-12T12:00:00Z => returns a highway A8 representation at the given time

If a query parameter is defined on a resource, define the API behavior in all cases including error situations. For example, if an id parameter on a feature resource is supported then define the semantics of the following query examples:

.../collections/highways/items/A8?id=A8 => should the request return \*true\* or \*the resource itself\*?

.../collections/highways/items/A8?id=A81 => should the request return \*false\* or '\*NULL\*' (assuming the id of A8 is not A81)?

Another example for a query parameter could be properties (which can also be combined with other parameters):

.../collections/highways/items/A8?properties=name,geometry => return the highway A8, but only the name and geometry attributes

.../collections/highways/items/A8?time=2019-02-12T12:00:00Z&properties=name,geometry => return the highway A8 at the given time, but only the name and geometry attributes

Use of the query string to select resources is highly resource specific and must be described on a case by case basis.

6.3.7. Principle #7 – Error Handling and use of HTTP Status Codes

**Note: Error Codes are the developer’s insight into your API. So be precise and as detailed as possible. Inconsistent or inaccurate or ambiguous error handling is often one of the biggest complaints when using a Web API.**

Associate each error situation of the API with the appropriate HTTP status code (see also Principle #8).

However, also consider supporting a "switch off" that always returns a status code 200 plus additional (debug / insight) information in the HTTP response body

e.g. ?suppress\_response\_codes=true

Return detailed human readable error no. + description  
information on how to fix things + contact details

{ "developer\_message": "…",

"user\_message": "...",

"error\_code": "...",

"contact\_details": "..."

}

6.3.8. Principle #8 – Use of HTTP Status Codes

More than 70 HTTP status codes exist (summary is in RFC 7231). You should reduce the use in your API to the most important ones. For example:

| **Option Set #1 – Basic set** | **Option Set #2 – Additional** |
| --- | --- |
| - 100 - Continue |  |
| - 200 - OK | - 201 - Created |
|  | - 204 - No Content |
|  | - 304 - Not Modified |
| - 400 - Bad Request | - 401 Unauthorized |
|  | - 403 - Forbidden |
|  | - 404 - Not Found |
|  | - 405 - Method Not Allowed |
|  | - 406 - Not Acceptable |
|  | - 409 - Conflict |
|  | - 410 - Gone |
|  | - 412 - Precondition Failed |
|  | - 415 - Unsupported Media Type |
|  | - 422 - Unprocessable Entity |
|  | - 429 - Too Many Requests |
| - 500 - Internal Server Error | - 503 - Service Unavailable |

Be explicit which 30x status codes (Redirects ( 300 – 399 ) the API supports. For any supported 30x codes follow the HTTP semantics.

6.3.9. Principle #9 – Use of HTTP Header

Define all HTTP Headers that the API supports.

Use HTTP Headers as specified in RFC 2616, but design the API to allow overwriting of HTTP Headers based on URL query parameters.

For support of caching, consider supporting entity tags and the associated headers. However, their use might be in conflict when implementing security requirements. For these cases, explicitly name those headers that must be overwritten to avoid caching.

6.3.10. Principle #10 - Content Negotiation

In HTTP, content negotiation is the mechanism that is used for serving different representations of a resource at the same URI, so that the user agent can specify which is best suited for the user (for example, which language of a document, which image format, or which content encoding). (Mozilla, 2021)

Content negotiation is an important aspect of any Web API specification and implementation but special case of Principle #9.

The primary mechanisms for HTTP based content negotiation are these request headers:

* **Accept:** Which media types are acceptable for the response, such as "application/json," "application/xml," or a custom media type such as "application/vnd.example+xml"
* **Accept-Charset:** Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
* **Accept-Encoding:** Which content encodings are acceptable, such as gzip.
* **Accept-Language:** The preferred natural language, such as "en-us".

In OGC Web API specification development, use an HTTP request header such as 'Accept' or 'Accept-Language' to request the response in a particular content type or language as defined in [RFC 2616](https://tools.ietf.org/html/rfc2616).

Use registered [IANA Media Types](https://www.iana.org/assignments/media-types/media-types.xhtml) whenever possible.

An example for content negotiation based on HTTP headers and with query parameter override:

HTTP 1.1 GET .../collections/highways/items/A8

accept: application/geo+json

=> should return the response using the GeoJSON encoding

HTTP 1.1 GET .../collections/highways/items/A8?accept=application%2Fgml%2Bxml

accept: application/json

=> should return the response using the GML encoding

6.3.11. Principle #11 - Pagination

OGC APIs that may be designed to access large data collections should support pagination. *Offset* pagination is one of the simplest to implement. Offset pagination is specified using the limit and offset commands. Offset pagination is popular with apps powered by SQL databases, as limit and offset are already included with the SQL SELECT library. Offset pagination requires almost no programming. It’s also [stateless](https://nordicapis.com/defining-stateful-vs-stateless-web-services/) on the server side and works regardless of custom sort\_by parameters.

The downside of offset pagination is potential performance difficulties when dealing with large offset values.

As an example, use **limit** and **offset** as "query-string" parameters:

.../collections/highways/items?limit=50&offset=101 => returns up to 50 highways starting at position 101

The OGC defined API should return metadata with each response providing the total number of resources available (e.g. total) in the payload as well as the link to the next page.

As a supplement consider support for Web Linking (RFC 5988)

– Use HTTP Response Header to provide URLs for fetching the next / previous page – This approach is application neutral and should be provided by the API as the default

6.3.12. Principle #12 – Processing Resources

Use **verbs** to offer **operations** on resources. Verbs specify an action to be performed on a specific resource or a collection of resources For example:

.../transform => represents a processing resource that specifies transforming a resource

The parameters of the process are provided as query parameters. For example:

.../transform?in=.../collections/highways/items/A8&toCRS=http://www.opengis.net/def/crs/EPSG/0/3258 => returns the A8 highway in the coordinate reference system ETRS89 lat/long

Note that the result of the example above may result in the same response as a selection/negotiation parameter on the resource (see Principle #6). For example:

.../collections/highways/items/A8?crs=http://www.opengis.net/def/crs/EPSG/0/3258

An OGC API standard may allow offering processing resources as separate operations to support an explicit separation and highlight the processing capability. This allows publishing explicit metadata about the process, such as the input and output data structures.

6.3.13. Principle #13 – Support Metadata

Providing metadata support as part of an OGC Web API specification helps the both the specification and implementation developer to understand how to use data or processing resources supported in the API specification. Two approaches exist for achieving this:

(1) Start the URL path with 'metadata' to indicate that subsequent path identifies a resource for which the metadata is returned.

.../metadata/collections/highways/items/A8

(2) End the URL path with 'metadata' to indicate that the metadata is an integral part of the resource that can be fetched separately.

.../collections/highways/items/A8/metadata

Regardless of the approach taken, use it consistently.

You may use of the ‘?’ operator to send selection criteria (see Principle #6).

6.3.14. Principle #14 – Consider your Security needs

Try to follow common practices for security in IGC Web APIs. For example:

* Provide access to the OGC API using HTTPS.
* Include support for authentication from the beginning.
* Consider consistent support for CRUD (Create, Read, Update, Delete) from the beginning (see Principle #5);
* Support for Execute may be provided on processing resources (see Principle #12) or using POST (see Principle #5).

6.3.15. Principle #15 – API Description

Describing the OGC API in human and machine readable form has value to the developer. Currently the OGC is using OpenAPI version 3 as common practice for documenting an OGC API.

6.3.16. Principle #16 - Use IANA well-known identifiers

IANA and other standardization organizations have defined so called well known identifiers for different purposes. For example:

* Media types: <https://www.iana.org/assignments/media-types/media-types.xhtml>
* Link relations: <https://www.iana.org/assignments/link-relations/link-relations.xhtml>
* Well-known URIs: <https://www.iana.org/assignments/well-known-uris/well-known-uris.xhtml>

For example, is it possible to differentiate between XACML or GeoXACML policies? XACML policies would be returned with the 'application/xacml+xml' media type and GeoXACML policies with media type 'application/geoxacml+xml'.

6.3.17. Principle #17 - Use explicit geospatial relations

In many cases using typed relation to explicitly declare links among resources is appropriate. In the geospatial world, a special case are the possible topological spatial relations between resources (e.g., contains, within, etc.) which are easy to determine using most geospatial technology (GIS) software, but not necessarily with Web clients unless the relations are explicitly represented. The relations may either be explicitly included in the resource representation or in Link headers in the HTTP response header (see RFC 5988).

6.3.18. Principle #18 - Support W3C Cross-Origin Resource Sharing

If the OGC Web API is designed to be accessed by Web-applications executed in a Web Browser, support W3C CORS (<https://www.w3.org/TR/cors/>).

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources. (Mozilla, 2021)

This approach provides the ability to overcome the security restrictions introduced by the Same-Origin Policy (<https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy>) applied by the Web Browser to JavaScript based applications when trying to access your Web API.

As identified in W3C CORS, in cross origin cases the HTTP request carries specific HTTP headers and it is expected by the Web Browser that associated HTTP response headers exist in the response. Otherwise the processing stops.

An example of a cross-origin request: the front-end JavaScript code served from https://domain-a.com uses [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) to make a request for https://domain-b.com/data.json.

6.3.19. Principle #19 - Resource encodings

The API should provide resource representations based on the expectations of the developers.

When designing an OGC API, a key decision is whether or not the Web API should support a default encoding that every implementation has to support. The SWG should recommend supporting JSON and HTML as encodings for all resources. JSON is recommended as it is a commonly used format that is simple to understand and well supported by tools and software libraries. HTML is recommended as it is the standard encoding for Web content.

Finally an XML encoding should be supported as it is often required to meet specific security requirements. Also, many existing IT standards and OGC encodings are based on XML.

6.3.20. Principle #20 - Good APIs are testable from the beginning

Any OGC Web API developed according to these guidelines can be tested during its prototyping phase. Considering all design principles including the identification of resource types, the effect of applying HTTP methods to them, the potential HTTP status codes, etc. provides the basis for documenting and implementing compliance tests in parallel with the API design and prototyping.

Annex A: OGC Web API Guidelines

Each Standards Working Group (SWG) shall complete this checklist when the SWG submits a standard to the OAB for consideration ahead of the Public Request for Comment (RFC).

|  |  |  |
| --- | --- | --- |
| **#** | **Principle** | **If and How the principle is met by the candidate standard** |
| 1 | Don’t Reinvent |  |
| 2 | Keep It Simple and Intuitive |  |
| 3 | Use Well-Known Resource Types |  |
| 4 | Construct consistent URIs |  |
| 5 | Use HTTP Methods consistent with RFC 2616 |  |
| 6 | Put Selection Criteria behind the ‘?’ |  |
| 7 | Error Handling and use of HTTP Status Codes |  |
| 8 | Use of HTTP Status Codes |  |
| 9 | Use of HTTP Header |  |
| 10 | Content Negotiation |  |
| 11 | Pagination |  |
| 12 | Processing Resources |  |
| 13 | Support Metadata |  |
| 14 | Consider your Security needs |  |
| 15 | API Description |  |
| 16 | Use IANA well-known identifiers |  |
| 17 | Use explicit geospatial relations |  |
| 18 | Support W3C Cross-Origin Resource Sharing |  |
| 19 | Resource encodings |  |
| 20 | Good APIs are testable from the beginning |  |