API versioning

The term API version refers to what is written in the version field of an API OAS definition file.

API VERSIONING IN A NUTSHELL

API versioning is done using SemVer 2.0.0: vx.y.z with x, y and z being numbers indicating MAJOR, MINOR and PATCH versions.

In addition to public-release API versions x.y.z, internal to the CAMARA project during the API release cycle, version extensions are used for pre-releases (described in the section below):

- alpha API versions
- release-candidate API versions

Finally,

- work-in-progress (wip) is used as the API version before the first pre-release and between 2 pre-releases
- as an exception, initial API versions (APIs with MAJOR version x = 0) may have both the MAJOR and the MINOR version number (v0.y), separated by a dot ("."), in the API version in the URL, e.g. {apiRoot}/qod/v0.10

The following types of API versions are used throughout an API's release cycle:

- **work-in-progress (wip)** API versions used before the first pre-release or between pre-releases. Having "wip" in the API version field indicates that PRs are ongoing and that the API is not usable by API consumers. The first PR should set the API version to wip, and the next "release PR" shall set the API version to the next (semver) x.y.z version.
- **alpha** (x.y.z-alpha.m) API versions (with extensions) for CAMARA internal API rapid development purposes
- **release-candidate** (x.y.z-rc.n) API versions (with extensions) for CAMARA internal API release bug fixing purposes
- **public-release** (x.y.z) API versions for publication as part of a meta-release. These API versions only have API version number x.y.z (semver 2.0), no extension. The public-release API can have one of two maturity states:
  - **initial** - indicating that the API is still not fully stable (x=0)
  - **stable** - indicating that the API has reached a certain level of maturity (x>0)

The use of the different API versions throughout the API release cycle is illustrated in the following figure.

API versions throughout the release process

```
  wip (develop)  wip (test)
  |         |         |
  |         |         |
  |         |         |
  alpha    release-candidate
  |         |         |
  |         |         |
  |         |         |
  M3       M4
  |         |         |
  |         |         |
  M0       public-release
  |         |         |
  |         |         |
  |         |         |
  wip (patch)
```

The next sections provide the definition of terms used and the description of the different API version types when moving through the release cycle.

* Definitions
Definitions

API version

The API version is defined in the "version" field of the OAS definition file of an API. Its content must follow Semantic Versioning 2.0.0 | Semantic Versioning (semver.org).

An API version has the format: x.y.z where x, y and z are numbers corresponding to MAJOR, MINOR and PATCH versions.

The following bullets are extracted from the semver 2.0 specification (and links to the corresponding spec item):

- Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable. https://semver.org/#spec-item-4
- Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes. https://semver.org/#spec-item-5
  - Patch version Z (x.y.Z | x > 0) MUST be incremented if only backward compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior. https://semver.org/#spec-item-6
  - Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backward compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
  - Major version X (X.y.z | X > 0) MUST be incremented if any backward incompatible changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented.
- Note: in CAMARA v1.0.0 is considered the first stable public API, while API versions v0.y.z can be public, but are considered unstable (initial).

Precedence example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1 < 3.0.0.

API version extensions

During API development and release preparation, API version extensions are used to identify intermediate versions of an API.

The API version extension is added to the "version" field of the OAS definition of an API, as well as to the URL (see below table). Its content must follow the here described syntax and semantics.

The following extensions may be used in the API version:

- alpha API versions: -alpha.m
- release-candidate API versions: -rc.n

Extensions are post-fixed to the API version numbers and separated from it by a hyphen "-". m and n are numbers starting at 1.

All extensions must have a number (so just -alpha or -rc is not allowed) and should be minimally -alpha.1 or -rc.1.

The very first version of an API for which a release can be created is 0.1.0-alpha.1.

API version in URL

The OAS API definition file contains the URL definition for the API endpoint which includes the version of the API with the following format:

- A lowercase "v" followed by the MAJOR number from the API version, e.g. for qod v2.1.0, it will be v2 as follows:
  ```yaml
  servers:
    url: {apiRoot}/qod/v2
  ```

- Once a stable public-release API version is created, the API version in the URL shall ONLY contain "vx", and never include an extension.

  **Exception:** For initial public-release API versions 0.y.z, the MINOR number may be included in the URL, e.g. v0.y. A dot ("."") is used for readability.
  - In this stage of the API development, y is used to indicate MAJOR (breaking) changes, and z is used to indicate MINOR (non-breaking) changes. Patches, if needed would also increase z.
  - For alpha and release-candidate API versions, the API version extension shall be included in the URL, but without any hyphens or dots.
    - Example: in the URL of an alpha API version 2.y.z-alpha.1, the version in the URL is v2alpha1.

API name
The API name is the segment in the `url` field in the OAS definition file before the segment holding the API version.

Example: for the above `url: {apiRoot}/qod/v2`, the API name is `qod`.

## API version types

The following tables provides the characteristics of the different API version types that an API will go through during its release cycle.

<table>
<thead>
<tr>
<th>API version type</th>
<th>Purpose &amp; characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-in-progress (wip)</td>
<td>The purpose of a work-in-progress API version is to indicate that the API is unstable due to one or more PRs being committed, possibly resulting in temporary inconsistencies.</td>
</tr>
<tr>
<td></td>
<td>• A work in progress (wip) API version has the following information in the API OAS definition file:</td>
</tr>
<tr>
<td></td>
<td>◦ Info object - version: wip (instead of x.y.z)</td>
</tr>
<tr>
<td></td>
<td>◦ Servers object - url: vwip (instead of vx)</td>
</tr>
<tr>
<td></td>
<td>• A wip API version may exist in 2 cases</td>
</tr>
<tr>
<td></td>
<td>◦ during the first introduction of a new API up to the creation of the first alpha API version</td>
</tr>
<tr>
<td></td>
<td>◦ during the time that one or more Pull Requests (PRs) are committed after a given release of an API.</td>
</tr>
<tr>
<td></td>
<td>• Between different PR commits, the API version remains &quot;wip&quot;.</td>
</tr>
<tr>
<td></td>
<td>• After or with the last PR commit, the API version and URL version are set to the next API version.</td>
</tr>
<tr>
<td></td>
<td>• The maintainers of the API decide on the release (or not) of the resulting next API version, implying that its corresponding release assets will be created.</td>
</tr>
<tr>
<td>alpha (alpha)</td>
<td>The purpose of an alpha API version is to support rapid development of an API and the creation of test implementations for feedback purposes.</td>
</tr>
<tr>
<td></td>
<td>• An alpha API version (x.y.z-alpha.m) is an intermediate API version with changes with respect to a previous API version.</td>
</tr>
<tr>
<td></td>
<td>• An alpha API version cannot be part of a CAMARA meta-release. It needs to go through the below release-candidate API version(s) API first.</td>
</tr>
<tr>
<td>release-candidate (rc)</td>
<td>The purpose of a release-candidate API version is to allow for (only) bug fixing encountered during further API implementation testing.</td>
</tr>
<tr>
<td></td>
<td>• A release-candidate API version (x.y.z-rc.n) is a pre-release of an API version which is stabilized and intended to become the next public-release API version.</td>
</tr>
<tr>
<td></td>
<td>• It shall be used to create API implementations and test the API and to create required release assets.</td>
</tr>
<tr>
<td></td>
<td>• A release-candidate API version can be proposed for meta-release. In this case, the public-release API version needs to be prepared.</td>
</tr>
<tr>
<td>public-release</td>
<td>The purpose of a public-release API version is that it can be used by organizations outside CAMARA in application development.</td>
</tr>
<tr>
<td>initial public-release</td>
<td>• A public-release API version (x.y.z) can be more or less mature:</td>
</tr>
<tr>
<td></td>
<td>◦ A public-release API version with x=0 is referred as an initial API version.</td>
</tr>
<tr>
<td></td>
<td>◦ A public-release API version with x&gt;0 is referred to as a stable API version.</td>
</tr>
<tr>
<td>stable public-release</td>
<td>Stable public-release API versions are API versions with x&gt;0 (x.y.z without version extension).</td>
</tr>
<tr>
<td></td>
<td>Stable public-release API versions are the ones recommended for use in commercial applications. The user can expect that subsequent API versions will be backward-compatible with the one they are using, unless explicitly announced otherwise.</td>
</tr>
</tbody>
</table>

## Creation of a public-release API version

To create a public-release version of an API the following steps need to be taken:

- First, using `wip` in the API version, develop your API until it is sufficiently stable to create a first alpha API version `0.1.0-alpha.1`
- Create one (or more) alpha API versions to stabilize the API
- Create the first release-candidate API version for the M3 milestone (stable proposed API version for testing)
- Create one (or more) release-candidate API versions between the M3 and the M4 milestone following feedback from testers
- Create the final release-candidate API version for the M4 milestone (stable proposed API version for public-release)
- Finally, create the public-release API version for public-release and, possibly, for inclusion in the meta-release

**IMPORTANT**
All alpha and release-candidate API versions are **INTERNAL to the CAMARA project and meant to be used only for the API development and release management process**. Usage for other purpose is at the user's own risk.

The below table gives an overview of the API versioning through the release cycle. The column headers mean the following:

- **API version type**: refers to the stage in the API release cycle: wip, alpha, release-candidate or public-release.
- **API version**: (without or with an extension) is what is put in the version field in the API OAS definition file.
  - NOTE: a public-release API version shall never have an extension.
- **API version in URL (initial/stable)**: is a lowercase "v" followed by the MAJOR number from the API version. This is put in the URL field in the Servers object in the API OAS definition file.
  - NOTE: for an initial API version (x=0), the API version in the URL is exceptionally allowed to contain both the MAJOR and the MINOR version numbers (v0.y).
- **API version can be released**: A release can be created for the API version (with or without (for alpha) a release package).

The following table gives the values of the API version (Info object) and the API version in the URL (servers object) used in the release process of an API version.

<table>
<thead>
<tr>
<th>API version types in the release process</th>
<th>API version (OAS object)</th>
<th>initial (x=0) API version in URL (OAS Servers object)</th>
<th>stable (x&gt;0) API version in URL (OAS Servers object)</th>
<th>API version can be released</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-in-progress</td>
<td>wip</td>
<td>vwip</td>
<td>vwip</td>
<td>No</td>
</tr>
<tr>
<td>alpha</td>
<td>x.y.z-alpha.m</td>
<td>v0.yalpham</td>
<td>vxalpham</td>
<td>Yes (internal)</td>
</tr>
<tr>
<td>release-candidate</td>
<td>x.y.z-rc.n</td>
<td>v0.yrcn</td>
<td>vxrcn</td>
<td>Yes (internal)</td>
</tr>
<tr>
<td>public-release</td>
<td>x.y.z</td>
<td>v0.y</td>
<td>vx</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Precedence examples for API versions throughout its release cycle:

- 0.1.0 < 0.2.0-alpha.1 < 0.2.0-rc.1 < 0.2.0-rc.2 < 0.2.0 (initial public-release)
- 1.0.0 < 1.1.0-alpha.1 < 1.1.0-rc.1 < 1.1.0-rc.2 < 1.1.0 (stable public-release)

**Update of a public-release API version**

Updates to a public-release API version can concern MAJOR, MINOR or PATCH changes to the API definition.

- For **MAJOR** or **MINOR** changes, the process is the same as for the creation of a public-API version (see previous section), going through alpha and release-candidate API versions to create the update API version.
- For **PATCH** changes, the API Sub Project can decide to either
  - apply the CAMARA API release process, as for MAJOR and MINOR changes resulting in a new API version x.y.z+1
    - 1.0.0 1.0.1-alpha.1 ... 1.0.1-alpha.m 1.0.1-rc.1 ... 1.0.1-rc.n 1.0.1 (next stable public-release)
    - apply a short cycle to create a maintenance-release of the API version.

It is recommended to have maximally 2 consecutive public-release API versions available at any given point in time.

**Breaking and non-breaking changes**

**Semantic versioning on breaking changes:**

- breaking changes imply a major version increment (that's the fundamental part)

What is a breaking change?

- deleting operations or actions on a resource. For example: POST requests on a resource are no longer accepted.
- adding a new mandatory input parameters. For example: now, to register a resource, a new required field must be sent in the body of the request.
- modifying or removing a mandatory parameter in existing operations (resource verbs). For example, when consulting a resource, a certain field is no longer returned, or a field that was previously a string is now numeric. (breaks existing payloads)
- modifying or removing an endpoint (breaks existing queries)
- changing input parameters from optional to mandatory. For example: when creating a Person resource, the age field, which was previously optional, is now mandatory.
- modifying
- adding new responses to existing operations. For example: creating a resource can return a 412 response code.

What is a non-breaking change?
• adding a new endpoint
• adding a new operation (POST, GET, …)
• changing a mandatory parameter into an optional one. For example: when creating a resource, a property of said resource that was previously mandatory becomes optional.
• adding optional input parameters to requests on existing resources. For example, adding a new filter parameter in a GET on a collection of resources.
• adding new properties in the representation of a resource returned by the server. For example, adding a new age field to a Person resource, which originally was made up of nationality and name.

How to avoid breaking changes (API provider) ?

• never change an endpoint name; instead, add a new one and mark the original one for deprecation in a minor release and remove it in a later major release (see semver FAQ entry: https://semver.org/#how-should-i-handle-deprecating-functionality)
• If possible, do the same for attributes
• new fields should always be added as optional.
• Postel's Law: "Be conservative in what you do, be liberal in what you accept from others". When you have input fields that need to be removed, mark them as unused, so they can be ignored.
• do not change the field’s semantics.
• do not change the field’s order.
• do not change the validation rules of the request fields to more restrictive ones.
• If you use collections that can be returned with no content, then answer with an empty collection and not null.
• layout pagination support from the start.

How to avoid breaking changes (API consumer) ?

• tolerant reader: if it does not recognize a field when faced with a response from a service, do not process it, but record it through the log (or resend it if applicable).
• ignore fields with null values.
• variable order rule: DO NOT rely on the order in which data appears in responses from the JSON service, unless the service explicitly specifies it.
• clients MUST NOT transmit personally identifiable information (PII) parameters in the URL. If necessary, use headers

Make the information available:

• provide an access to the new API version definition file (via a link or dedicated endpoint)
• if possible, do the same to obtain the currently implemented API version definition file