ONAP API Common Versioning Strategy (CVS) Guidelines

Executive Summary

General objectives of the ONAP Common Versioning Strategy (CVS) for all APIs:

- Implement semantic versioning (MAJOR.MINOR.PATCH) for APIs
- If necessary, refactor APIs to support the concept of MINOR releases; versioning scope and use cases provided
- Adopt a BWC policy for APIs that is current MAJOR release minus 1 year (to be re-visited post-Casablanca)

HTTP/REST API specific rules/policies:

- Implement pre-defined custom headers to communicate MINOR, PATCH, and LATEST VERSION
- Align URLs to include only the MAJOR version
- Documentation using Swagger 2.0, by following the guidance provided

ONAP CVS Proposal Deck was presented to the Architecture Committee on 4/3/2018, and modified on 4/11/2018 to correct the flow diagrams and case of the custom header names.

Clarification on terminology:

- **SHALL** (mandatory) is used to indicate a requirement that is contractually binding, meaning it must be implemented, and its implementation verified.
- **SHOULD** (non-mandatory) is used to indicate a goal which must be addressed by the design team, but is not formally verified.

Policies for All APIs

Implementing Semantic Versioning for APIs

- Utilizes the same semantic versioning methodology that is being used for ONAP's Release Versioning Strategy; therefore, development teams are familiar with the definition of the methodology.
- For a given a version number, MAJOR.MINOR.PATCH, increment the:
  - MAJOR position when you make any incompatible API change
  - MINOR position when you add functionality in a backwards-compatible manner
  - PATCH (or BUILD) position when you make invisible (and thus backwards-compatible) bug fixes
- Details of the specification can be found at http://semver.org/

Versioning Scope and Use Cases

Definition of Layers (second column in the table below)

- API-wide - versioning at this level creates a new set of URIs to identify the resources
- Resource-wide - independently version different parts of the API's resource model; major change to a specific resource model with the notion that not all the resource models will change (namespace changes, add a new resource)
- Representation layer - modify and enhance the format, structure or content of resources that are served by the API without changing the resource model
- Behavior change - changes in behavior that do not change the resource model or representation; can include changes in process or state management

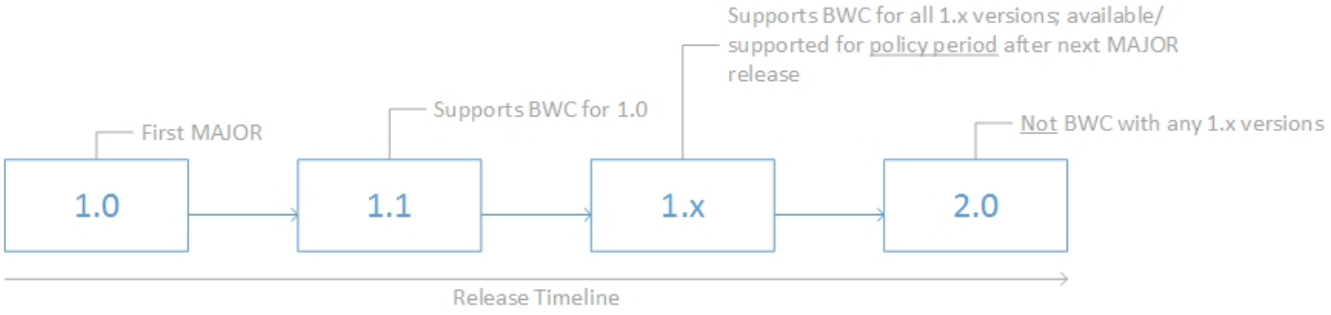
Use Case	Layer	MAJOR	MINOR	PATCH
				1

Refactoring resource model, if it has become fragile or overly complex through many evolutionary steps; introduce a set of namespaces to reflect the category of resources (nothing is where it used to be)	API	X		
Restructure resource(s) to meet new business requirements/conform to emerging interface standard(s)	API	X		
Add a required request parameter value without default	API	X		
Add a required request parameter value with default	API		X	
Add a new resource model or type	Resource		X	
Update an existing resource model or type w/BWC	Resource		X	
Update an existing resource model or type w/out BWC	Resource	X		
Adding optional data items to an input resource representation	Resource		X	
Add a new behavior method w/no changes to existing behavior methods (e.g. add PUT as a method when it did not exist as prior functionality)	Behavior		X	
Changes to data volume on returned response	Behavior		X	
Changes to undocumented sorting order of data on returned response w/out changes to volume	Behavior		X	
Changes to documented sorting order of data on returned response w/out changes to volume	Behavior	X		
Changes in behavior that do not change the resource model or representation	Behavior		X	
Deprecate method, but do not change the structure of the resource model or representation	Behavior	X		
Adding new optional parameter(s) (that do not change default behavior) to requests	Representation		X	
Adding data items to an output resource representation, where any prescribed schema validation (for example, XML Schema or JSON-Schema validation) is not broken	Representation		X	
Fix a defect that does not impact behavior or representation (e.g. fix internal algorithm to run more efficiently)	General			X
Changes to error codes, whereas the error code content is updated or changed, with no change in the resource model or representation	API		X	

[1] PATCH refers to the position in the version number, not the HTTP method of PATCH. This method should not be used as it is idempotent.

Backward Compatibility (BWC) Policy - TO BE REVISITED DURING OR POST-CASABLANCA

- API BWC shall be defined for **MAJOR releases as the current release - 1 year** (to be re-visited post-Casablanca). In other words, if an API is currently at 1.12 and a MAJOR release occurs to increment the version to 2.0, 1.12 (which is BWC for versions 1.0-1.11) must be functional /available for the period of 1 year after 2.0 is released.
- API owners shall ensure the previous MAJOR release remains available and functioning, in its last available production state, for the period of the BWC policy.
- MINOR releases shall be not time or release-based, as they are assumed to be BWC.
- API owners shall ensure no end-to-end services break with the deprecation of an API, due to the BWC Policy. End-to-end services includes, but is not limited to, VNFs, PNFs, Networks, Allotted Resources, etc.



API Evolution and BWC

Policies for HTTP/REST APIs

API Custom Headers and Behavior

Custom Headers and General Rules

- Three custom headers shall support versioning in APIs:
  - *X-MinorVersion*

    - *X-PatchVersion*
    - *X-LatestVersion*
- The request from the client shall not break, if the headers are absent in the request.
- Clients shall ignore additional values from the payload in the response, if provided.
    - It shall be explicitly specified in the interface contract that a server may increment a MINOR version and add additional fields.
    - The client shall be capable of handling this type of change in contract, if they remain on a previous MINOR version.
- The server shall employ logic to fallback to the **MAJOR** version of the API, in the event that *X-MinorVersion* is not provided (see use case below).

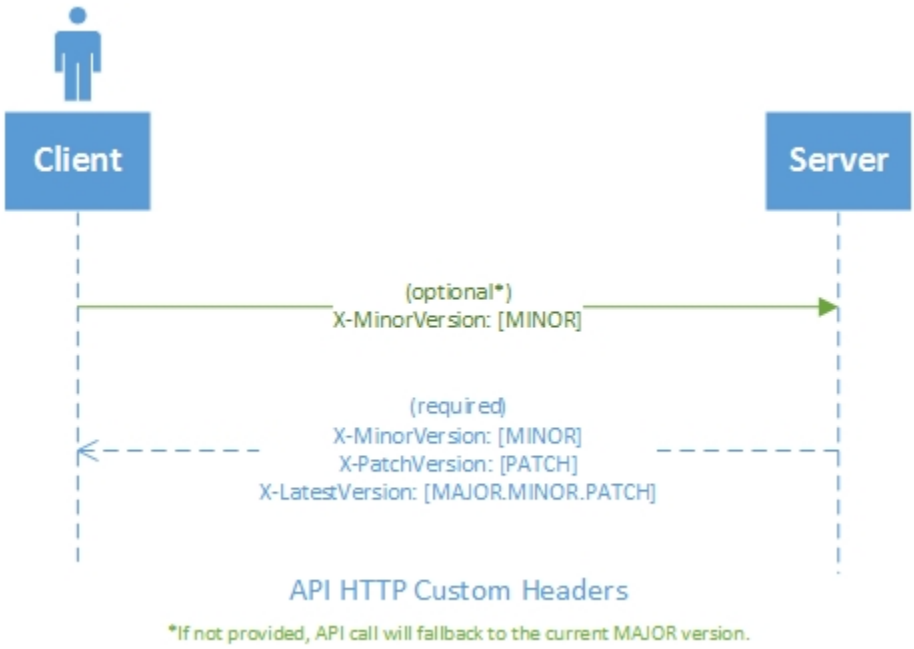
Use case for falling back versus failing forward to the MAJOR version of the API

The vserver entity in v1 of ECOMP had no “prov-status” field.
The prov-status field was added in v1.1 as a non breaking change.

The RO client PUTs all the data in the vserver except the prov-status field, so they use v1.
The GFP client manages the prov-status field in the vserver. They use v1.1.

A REST PUT must include the entire representation of the object.
Therefore a v1 PUT does not include the prov-status but the v1.1 PUT must.

If only major versions get passed, and the system should fail forward, a PUT by RO lacking the prov-status field would wipe out the prov-status value.

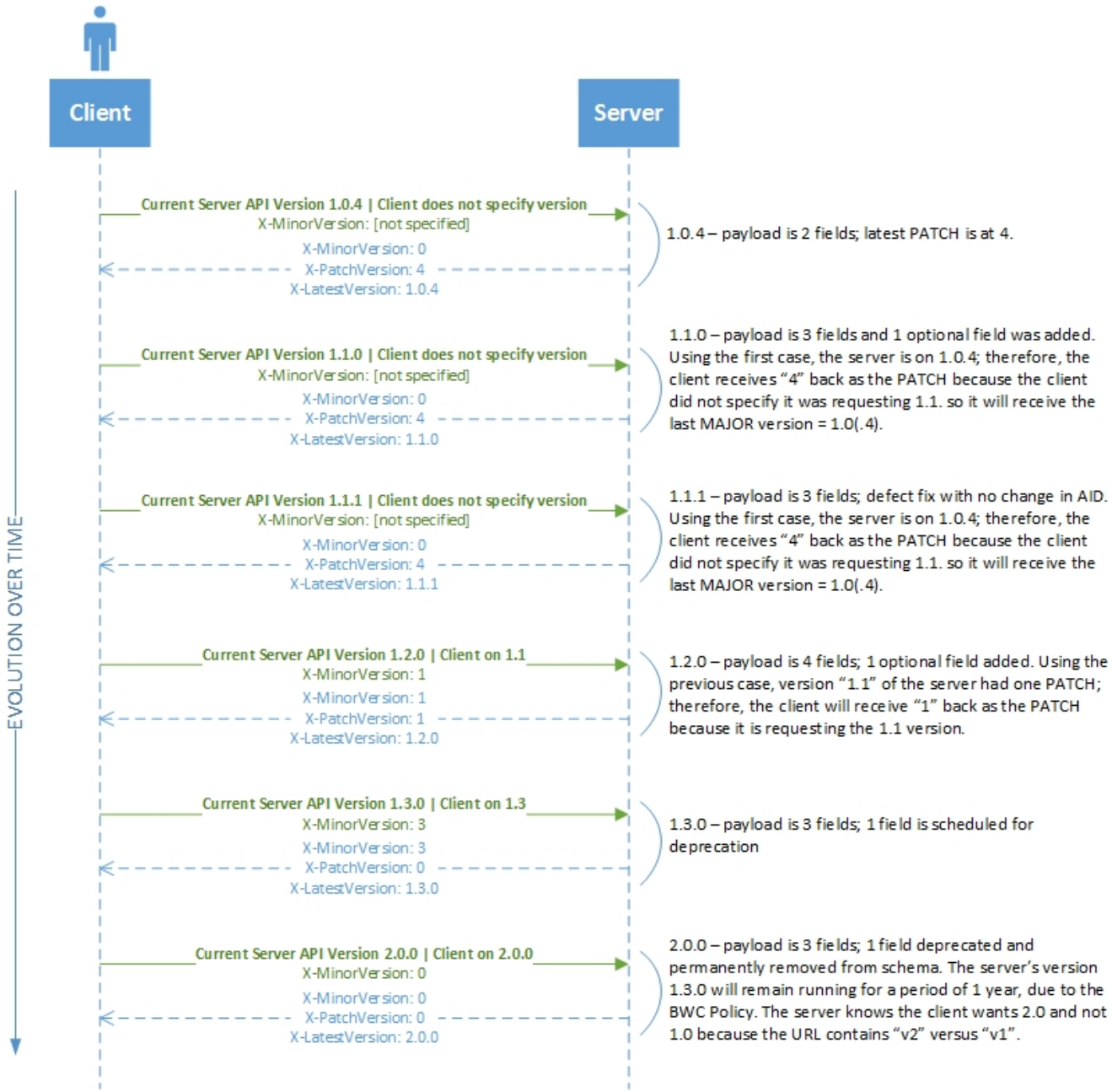


Custom Headers Specification

Header Name	Specification
X-MinorVersion	• Used to request or communicate a MINOR version back from the client to the server, and from the server back to the client • This will be the MINOR version requested by the client, or the MINOR version of the last MAJOR version (if not specified by the client on the request) • Clarification: This will always be the MINOR version requested by the client - OR - if the client does not specify, it will default back to the very first MAJOR version of the server. For example, if the server is on 1.1 and the client does not send *X-MinorVersion*, the API call will default to 1.0 which makes the MINOR version = 0. This lets the client know they are not receiving the latest version, and they will know because *X-LatestVersion* will notify them. • Contains a single position value (e.g. if the full version is 1.24.5, *X-MinorVersion* = "24") • Is optional for the client on request; however, this header should be provided if the client needs to take advantage of MINOR incremented version functionality • Is mandatory for the server on response

X-PatchVersion	• Used only to communicate a PATCH version in a response for troubleshooting purposes only, and will not be provided by the client on request • This will be the latest PATCH version of the MINOR requested by the client, or the latest PATCH version of the MAJOR (if not specified by the client on the request) • Clarification: This will always be the PATCH version the server is running. • Contains a single position value (e.g. if the full version is 1.24.5, *X-PatchVersion* = "5") • Is mandatory for the server on response
X-LatestVersion	• Used only to communicate an API's latest version • Is mandatory for the server on response, and shall include the entire version of the API (e.g. if the full version is 1.24.5, *X-LatestVersion* = "1.24.5") • Used in the response to inform clients that they are not using the latest version of the API

Custom Header Flow Diagrams



URL Structure Policy

- The URL shall only contain the **MAJOR** version number to minimize changes in the URL for MINOR and PATCH releases, assuming MINOR. PATCH releases are BWC.
- The structure of the URL shall be as follows, where version is placed after the "service" or API name:

  .../root/{service or API name}/v{version number}/{resource path}

  Example: {hostname}/aai/resource/v14/complexes

  *Note: "v" should precede the MAJOR version number in the URL. Service or API name is not the resource; it is intended to group of set of related resources.

FOR RESTCONF APIs ONLY

In RESTCONF, APIs are organized by "modules", which for our purposes we can say are analogous to services. There are 3 different types of APIs, each with its own standard URI format:

- Configuration data (also referred to as "config tree"), which stores the in flight view of network data (so it shows pending changes).
  - Required URI is /restconf/config/{module}/{resource}
- Operational data (also referred to as the operational tree), which stores the current active view of network data.
  - Required URI is /restconf/operational/{module}/{resource}
- RPCs
  - Required URI is /restconf/operational/{module}/{rpc name}

URIs for ONAP will follow the convention below:

- Configuration data: */restconf/config/{service}:v{version}_{resource}* *e.g. /restconf/config/neutron:v2_networks*
- Operational tree : */restconf/operational/{service}:v{version}_{resource} e.g. /restconf/operational/neutron:v2_networks*
- RPC : */restconf/operations/{service}:v{version}_{rpc} e.g. /restconf/operations/SLI-API:v1_healthcheck*

Requirements/Extensions of the Swagger 2.0 Specification

Spec ID	Specification/Requirement
SG-1	All components shall use Swagger 2.0. The specification may be found here. OpenAPI 3.0 is a roadmap item.
SG-2	Within the **Info Object**, the following annotations **are included** in the Swagger specification and shall be **required**, even if they are optional in the Swagger spec: **title** **description** **version** - fully-qualified version number of the Swagger file (ex: 1.4.18)
SG-3	Within the **Info Object**, the following **are extensions** of the Swagger specification and shall be **required**: **x-planned-retirement-date** - use YYMM; string type. *This is the date that the API shall be deprecated, based on the BWC Policy. NOTE : APIs may be active after their retirement date, but are not guaranteed to remain in production. An API retirement may be pushed out to accommodate BWC for clients.* **x-component** - SDC, MSO, SNIRO, etc., or the mS name; string type. *This is the component that primarily owns the API from a development perspective.*
SG-4	Under the **Path**, the following shall be **required**: **x-interface info** - this contains two attributes: ▪ **api-version** - fully-qualified version number of the API (ex: 1.3.6); string type. *This is the version of the API. This differs from the version in SG-2 above. Components shall follow the Versioning Use Cases above to determine how to evolve API versions.* ▪ **last-mod-release** - use release number or name (this should be consistent, choose either one); string type. *This is the last release that the API was modified in.*
SG-5	Swagger files shall be generated at build time, and be placed in a centralized ReadTheDocs repository: http://docs.onap.org

SG-6	Within the **Path Item Object**, the following are included in the Swagger specification and shall be **required**: **description** - string **parameters** ▪ **required** - boolean ▪ **type** - string

Working Team Information/Discussion

Working Team Members

Name	Company	Email	Phone Number	Time Zone
Rich Bennett	AT&T			
Dana Bobko*	AT&T	dw2049@att.com	(561) 371-7619	EST
Sharon Chisholm	AMDOCS	sharon.chisholm@amdocs.com		EST
Chris Donley	Huawei	Christopher.Donley@huawei.com		
Gregory Glover	AT&T	gg2147@att.com	(847) 420-8459	
Mark Ho	AT&T	mh574f@att.com	(781) 791-4345	EST
Ramki Krishnan	VMware	ramkik@vmware.com		
Andy Mayer	AT&T	am803u@att.com		EST
Adolfo Perez-Duran	ARM (OAM)	adolfo.perez-duran@oamtechnologies.com	720.560.2659	MT
Alexander Vul	Intel	alex.vul@intel.com		
Parviz Yegani	Huawei	Parviz.Yegani@huawei.com	(408) 759-1973	PT
HuabingZhao	ZTE	mailto:zhao.huabing@zte.com.cn		GMT+8

* Responsible team lead

Discussion Items

Item	What	Notes
1	R3 Focus /Scope	**Establish/finalize a proposal for a generic versioning methodology, URL structure for HTTP/REST APIs, and Swagger 2.0 /OpenAPI 3.0 guidance.** • The items in this scope are low hanging fruit that could be achievable for Casablanca; assess doability *after* the proposal is put forth in the community. • If one of the identified scope items for R3 cannot be achieved, it is assumed it will move into R4. • Dana will pull out all relevant items in her deck and park on this page below for the team to review. • The definition of MAJOR.MINOR.PATCH for the Semantic Versioning 2.0.0 specification is **very explicit**; we should not deviate from the definitions in the specification (like re-purposing the positions to hold another value). Current recommendations (on the table): • Utilize the semantic versioning methodology for APIs (MAJOR.MINOR.PATCH); same definition of the methodology being used for ONAP releases. • Provide use cases as guidance for incrementing version numbers (see below). • Provide URL structure policy (see below). • Provide requirements/extensions of the Swagger 2.0/OpenAPI specification (see below).
2	R4 and Beyond	**Establish/finalize a proposal for backwards compatibility (BWC) and exposing API versions.** • BWC would come *after* all the APIs are "speaking the same language" in how versions are characterized. Once that occurs, we can look to how those API versions are exposed to clients/within interfaces. • Dana Bobko has a proposal in her deck that talks about custom headers, but that is just one of many ways this could be done - plus, we need to consider if that will work for *every* API.
3	Noteworthy	Dana Bobko will be working with Gregory Glover to combine the results of this working team with the Documentation effort (there are intersection points).

4	Open issue	How do we deal with restconf interfaces (e.g. ODL-based components)? See Dan's comment

Resources/Related Links

1. REST APIs Must be Hypertext-Driven: Blog post where Roy Fielding argues that not any HTTP-based interface is a REST API
2. RESTful API Design Specification (for ONAP)
3. REST APIs don't need a versioning strategy – they need a change strategy