Integration Services

**July 2019**

Document Control Information

Document Information

|  |  |
| --- | --- |
| Document Identification | Integration Services  |
| Document Name | Services Versioning Strategy |
| Document Author | Darrell Cook |

Document Edit History

| Version | Date | Additions/Modifications | Prepared/Revised by |
| --- | --- | --- | --- |
| 0.1 | Oct 26, 2016 | Initial draft – with content (Overview, Strategy, Versioning Rules, Deprication) | Darrell Cook |
| 0.2 | Jan 19,2017 |  | Marissa Kitz |
| 0.3 | July 16, 2019 | Updated to include modifications made in standards document and correct table of contents and formatting issues | Christine Viele |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction and SOA Service Version Strategy Overview 4](#_Toc14184417)

[2 Service Versioning Rules 5](#_Toc14184418)

[3 Service Deprecation 6](#_Toc14184419)

[4 Service Packaging and Deployment 7](#_Toc14184420)

# Introduction and SOA Service Version Strategy Overview

This document is intended to provide the necessary and appropriate level of details around the standards and guidelines to be used in service versioning here at Yale. As part of the Workday initiative (R4), significant effort is underway to design, build and expose services accessing the People Hub data source. This would provide applications with a larger set of people data and populations types than is contained in Workday alone. This is Yale’s first attempt at delivering a true set of SOA (Service Oriented Architecture) services. Several artifacts are being created to provide the necessary framework for properly rolling out and implementing SOA services here at Yale. Such artifacts include a Service Contract, an Invocation Outcome standards document, a Web Service Security document, Consumer Usage Plans, as well as this Service Versioning strategy document.

With the creation of SOA services, reusability is a key principle. As more and more service consumers

are engaged, new requirements necessitate new service versions. Services will continuously need to

evolve and procedures must be in place to ensure existing consumers are not impacted. The Service

Versioning Strategy document describes what types of changes constitute major version changes versus minor version changes as well as guidelines for how Yale can best manage version deprecation to ensure that services are and can be properly retired. This is important so that service maintenance activities do not become unmanageable.

A well-documented and understood version strategy will help in dealing with changes to services, now and in the future. Versioning will allow for several variations of the same service to exist simultaneously. While ensuring that existing service consumers can continue to use older versions, it will enable newer consumers to move to any newer and more recent services available. The following illustration shows how two service consumers can consume different major service versions (and also choose whether to consume enhancements related to minor version changes as well).

Service Version 2

Service Version 1.1

Service Version 1.0

Service Consumer 2

Service Consumer 1

Can migrate easily to minor version with none to minimal impact

# Service Versioning Rules

The following section provides some of the rules / guidelines for changes that will ultimately determine whether a service will require either minor version changes or major version changes. Major versions will be noted in the subsequent document as M (M=Major version) and minor versions will be noted as m (m=minor versions).

Version routing will be used to support two versions of a service that will need to be available at the same time. The decision of how to route is based on the version of the Layer 7 gateway URL consumed.

There may be several influencing factors that can and will influence service changes. Areas for consideration include:

**Schema Changes**

There are areas where service schema changes will either constitute a minor version change or a Major version change.

The following changes made (or anticipated) to the service schema, that do not affect any existing consumers, would constitute **minor (m) version changes**.

* Insignificant changes to schema that would include adding documentation, formatting (not data type changes), commenting, etc. should not result in a major version
* If a service is not yet released in production and continues to undergo development changes. Major service versions should be done once they are deployed to production.
* Adding additional response attributes to the service that don’t have critical impacts to the overall schema

The following however are more intrusive changes and would have possible or definite service consumer impacts. These changes would constitute **Major (M) version changes**.

* The addition of new required elements to a type
* Changing of a request parameter from “optional” and/or “conditional” to “required”. Even relocation of elements within types from one part of the service to another.
* Adding or removing an enumeration value
* Removing or renaming a global type or element
* Any change to field data types or the addition of restrictions

**Production Fixes**

Production fixes are also another influencing factor and important aspect to service versioning. It is expected that a production fix would not result in a major version. It would be considered a fix to the current version (M.m) of the service that was previously deployed in production. Once deployed, a production fix to an existing service would increment the minor version by 1 (i.e. M.m+1).

**Other**

There may be other influencing factors or decisions that could also alter service versioning. Changes to the overall service implementation, back-out plans, etc. are other possible examples. Regardless of the change there should be proper analysis and agreement on whether the changes being made would constitute a major or minor version change. If agreement is not reachable, then it is always safest to initiate a major (M.) version change as well as proper communication to service consumers.

It is important to note that regardless of a minor version change or a major version change, existing consumers should be notified and proper testing should be performed to ensure that there are no impacts.

# Service Deprecation

The service deprecation (or sunsetting) strategy is an important aspect in order to support the delivery of new versions, while at the same time, maintaining existing versions. The service sunsetting strategy will primarily apply to major service versions. Minor service versions should not constitute service consumer changes and impacts, therefore will be implemented in production without impacts to the gateway URL’s.

The goal should be to support no more than three service versions (for a given service). These versions should be reflective of the following

* vM(n-1): This represents one service version back from the latest and greatest version in production
* vM(n): This represents the current service version in production
* vM(n+1): This would represent the future service release targeted for production based on new requirement changes. Prior to releasing another version of this service, enforcement on service consumers using the existing VM(n-1) should be made to move them to the newest version in production or ideally the future state version (to be tested along the same production deployment timeline).

# Service Packaging and Deployment

**\*\*\*Before beginning service updates, the service contract should be updated to reflect these changes.\*\*\***

1. Update the XSD for the request schema and response schema in the Git Repo (<https://git.yale.edu/BSG-SOA/Talend-Public>)
2. The name format is like this- ServiceName\_Request/Response\_v#.xsd. Example: [PeopleService\_Request\_v1.xsd](https://git.yale.edu/BSG-SOA/Talend-Public/blob/Dev/PeopleService_Request_v1.xsd)
3. There are multiple branches for Dev, Test, and Master (Production). The Talend and Layer7 jobs read from these files directly.
4. Make appropriate changes and change file name if required. For information on using Git visit this page (<https://isa.its.yale.edu/confluence/display/BSG/GitHub>)
5. Commit to Git Repo
6. Open Talend Job and make appropriate changes
7. Add into the comments version notes
8. If appropriate change the Version number fixed value in the XML output component
9. Publish as a Snapshot or release. For information on publishing from Talend visit this page ([https://isa.its.yale.edu/confluence/display/BSG/Talend+Web+Services+Guide](https://isa.its.yale.edu/confluence/display/BSG/Talend%2BWeb%2BServices%2BGuide))
10. Deploy compiled job with Jenkins (Development deploy to be done by Dev, Test by Management)
11. Follow information published at [https://isa.its.yale.edu/confluence/display/BSG/Jenkins+-+Deployments](https://isa.its.yale.edu/confluence/display/BSG/Jenkins%2B-%2BDeployments)
12. Open Layer7 Policy Manager and make any appropriate changes to the policy. (Development to be done by Layer7 tech contact)
13. Follow information published at <https://isa.its.yale.edu/confluence/display/BSG/Layer7>
14. Update version notes and documentation on BSG Public confluence site. The information updated there will auto-populate into the private site as well. (See pages titled "Version Notes" on this site [BSG Application Services Public Home](https://yaleits.atlassian.net/wiki/spaces/BASP/overview))
15. Have testing validated against Development for your changes before migration to test. Testing should be done by more then one person. Developer should also run through general testing to validate changes are working.