

Pilot API Versioning

Recommendations

Pilot Flying J

v2.0, 2023-11-17

# Overview

An API should be designed for the long-term, but since change is inevitable, the API will never be completely stable. API change management can be addressed by a comprehensive versioning strategy. The strategy should include a periodic API deprecation and retirement schedule, an up-to-date documentation describing the technical changes per version, and the business requirement necessitating the change.

For simplicity, API versions should be captured at a major version level. For each API version, multiple Exchange *asset* versions can be published but the API contract itself should only be managed at major versions. The below API versioning guidelines provide recommendations on what would constitute a version change.

# API Versioning Guidelines

## Supporting/Breaking Backward Compatibility

Not all changes require a new version. API versioning strategies can be categorized into two categories: backward-compatible and non-backward-compatible (or “breaking”).

Small changes that are backwards-compatible typically do not require a new version. Since a backwards-compatible change does not change the behavior of the existing API (even though a new resource has been added or the data types supplied by the API request/response have been expanded), the API can utilize the same major version as described. Once the initial major version has been deployed, newer ‘backward-compatible’ deployments can use the same major version.

Changes that are not backwards-compatible usually require the consumer to alter the way their clients interact with the API prior to interaction. Changes that are *not* backward-compatible (or “breaking”) should be well-documented so that potential consumers fully understand the differences between the API major versions. Breaking changes can be captured as a major version change (e.g. v1 to v2).

While the means of tracking changes is somewhat arbitrary, the central goal of versioning is to be consistent! Differing versioning strategies across the enterprise often lead to confusion for consumers and a reluctance to adopt APIs due to fear that breaking changes are introduced into a foundational API without requisite communication to existing consumers.

The table below describes scenarios for API versioning:

| **Type of Change** | **Major Version Change** | **No Version Change** |
| --- | --- | --- |
| **API Contract** | * Removing an existing API resource * Renaming an existing API resource * Changes in the resource hierarchy * Adding new *required* query parameters or headers * Renaming existing *required* query parameters or headers | * Adding a new API resource or operation * Adding new *optional* query parameters or headers |
| **Data Contract** | * Comprehensive changes in the data entity structure * Removing an existing data entity * Renaming an existing data entity | * Removing an existing data constraint * Add an optional data element or attribute |
| **Representation Format** | * Remove existing representation format (e.g. application/xml) | * Add a new representation format |
| **Security** | * Security model replacement * Renaming existing query parameters or headers used for security | * Add a new security model |

Renaming or modifying ‘optional’ query parameters/headers or data elements/attributes can be problematic due to usage. Some consumers might already be extensively using these ‘optional’ values in their requests. However, if no consumer is actively using these ‘optional’ values, then that might not require a version change.

## Special Considerations

Besides the types of changes listed in the table above, there are some special considerations when to introduce a new version or not:

* **Change in quality of service (QoS) -** When a change impacts the overall API quality of service (QoS), (i.e. API response time and availability), a new API version may be required as a negative change in the API QoS might break the applications that currently use the API.
* **Changing assumed domain of values in the output message** - Applications that use the API may have an expectation about the domain of values of certain fields in the response messages. This situation typically occurs when there is a change in the source system. For example, when moving from PeopleSoft to Workday, many of the fields will be similar, (e.g. Sick Days, Vacation Days etc.) but the values expressed within those fields may be different (e.g. the ways in which Sick/Vacation days are now calculated). This could have an adverse effect to existing consumers since they are expecting data to have a specific semantic meaning which is now different. In cases such as these, consider introducing a new API version.

# API Version Naming Conventions

* Specify the version with a "v" prefix.
* Use a simple ordinal number, for example, "v1","v2", and so on.
* Only specify the major version as part of the URL.

# API Version in Application URL

The API version should be featured in the application URL so that consumers can easily identify the version of the API they wish to call. There are a number of ways that the version can be specified in the application URL, but the following format is the most widely adopted:

<http://www.example.org/><api-name>/v1/<resource>

# Deprecation and Retirement

In general, avoid offering more than 3 versions *(n-3)* of the API at the same time. If a new version is being considered and 3 versions are already available, declare the oldest version of the API *deprecated*, update the developer’s portal to disallow any new requests for consumption, and notify all existing API consumers of the change.

Depending on the organization, provide a standard deprecation duration (e.g. 3 months) that will allow existing API consumers to implement and test against the new API and functionality before decommissioning/retiring the old API.

Once all consumers have moved to a newer API version or the deprecation duration has elapsed, notify the consumers that the API will be moved from *deprecated* to *retired*/*decommissioned*. If there are consumers that have not successfully migrated to a newer API version, work with the individual teams to devise a migration plan that will move them to a newer version.