Version Scheme and API Compatibility

For the purposes of understanding our versioning and API compatibility commitments we need to define what are considered parts of the Apache NiFi API. This following items are considered part of the NiFi API:

- Any code in the nifi-api module not clearly documented as unstable.
- Any part of the REST API not clearly documented as unstable.
- Any extension such as Processor, Controller Service, Reporting Task.
- Any specialized protocols or formats such as:
  - Site-to-site
  - Serialized Flow File
  - Flow File Repository
  - Content Repository
  - Provenance Repository
  - State management
- Any configuration file necessary to operate NiFi such as:
  - All files found in the nifi/conf directory
  - Templates
  - The configuration parameters and intended behavior of a given component/extension point

Anything not listed above is not considered part of the API and is subject to change in a manner that may differ from the guidance below. There are substantial portions of the codebase which are intentionally considered private from an API point of view. If there are specific items not listed here which should be considered part of the API please raise a discussion about it on dev@nifi.apache.org so it can be considered for inclusion. We want to keep the public surface of the Apache NiFi API well defined and well scoped so that those consuming the application either as a developer or user know what is considered safe and not safe to do but also so that our community can continue to rapidly evolve the codebase.

For the public API the Apache NiFi project aims to follow versioning principles as described at Semantic Versioning 2.0.0

Consider the following scenarios in the context of the most recent 'example' release being 0.0.1 and with the understanding that these are about the public API as defined above.

- For releases which are comprised solely of bug fixes or non-feature introducing or enhancing changes that requires only a 'patch' version bump (the Z part in X.Y.Z). So the next release then is 0.0.2.
- For releases which include backward compatible changes to introduce feature enhancements or new features that requires a 'minor' version change and the 'patch' version resets to '0' (the Y part in X.Y.0). So the next release then is 0.1.0. A 'minor' version change is also required for any change that could result in an existing flow becoming invalid, such as the addition of a required property with no default or the addition of a relationship, or the removal of a property or relationship. Note: it is **NOT** acceptable in a 'minor' version to change anything that can result in an existing flow behaving differently (other than a component becoming invalid). Doing so would fundamentally alter the way in which organizations process data without them realizing it.
- For releases which include non-backward compatible changes or changes deemed so substantive by the community that it is considered a 'major' version change and the minor and patch versions reset to '0' (the X part in X.0.0). So the next release then is 1.0.0.

After a release occurs the 'patch' version will be automatically adjusted by maven without the release manager doing anything special. So rarely will this value need to be manually set. In the event of a 'major' or 'minor' bump though the entire relevant source tree will need to be adjusted. That can be accomplished by using the following commands:

```
mvn versions:set "0.1.0-SNAPSHOT"
mvn versions:commit
```

Furthermore, we have defined the API above. But we must also be mindful of the effort required on users to move from major version to major version. We need to provide, to the extent possible, tooling to automatically migrate or uptake old configuration approaches. We need to ensure old protocols and new protocols can work together. We need to ensure that upgrades are as smooth and automatic as possible such that older versions configurations and data storage structures and formats are honored or have a clear conversion path.

> Special consideration: You must also consider 'compatibility' of processors and extensions specifically. For example, if a processor has two relationships and you add a third one or remove one you must be sure to take extra care to document this sort of thing for the release that it goes into. We should always provide a notice to folks upgrading from version to version so they know what to look out for. Those cases are usually easily resolved by simply knowing that after upgrading they need to specify a relationship and start that processor or that they need to remove a no longer relevant relationship. But the key is to ensure we follow through with effective notices for users so they understand what is happening and don't simply perceive a bug or issues introduced by upgrading.

Even within the concept of code compatibility, though, there can be some ambiguity. We should consider code to be 'backward compatible' (and therefore allowed in a 'minor' version change) if the change in X.Y.0 allows all extensions (Processors, Reporting Tasks, Controller Services, and Authority Providers) that are developed against the X.(Y-1).* API to still perform their tasks the same way. For example, it is not backward compatible to remove or change the signature of a method in the 'FlowFile' interface because Processors may no longer function properly. It is, however, acceptable to add a new method to this interface because it is expected that only the Framework will be implementing this interface. Therefore, all extensions will still function properly.

The community is also evaluating the use of more formalized mechanisms of communicating these concepts using approaches such as Apache Yetus Audience Annotations or built-in software development lifecycle models. If adopted this could allow us to do things like warn users and developers of components which they should not be extending from or which are unstable.

Given that NiFi itself is designed for extension and the growing community is providing frequent contributions in the form of feature enhancements and new features the most likely scenario in a given release is that it will be a 'minor' version bump.

It is not always clear what the right answer is for whether something is a 'patch' or 'minor' bump whereas 'major' should be pretty obvious. Changes to major should signal a potentially high risk but high reward scenario for would-be users. Changes to 'minor' and 'patch' though they should feel a relatively high degree of confidence in downloading and upgrading their system. That is something we'll have to earn as a community and it starts by following the guidance closely and adjusting it as we learn more. If unsure please initiate a dialog on the dev mailing list. This is really important information for those with commit privileges or who are reviewing contributions. Has the contributor properly accounted for the version change implications? If someone is providing a new feature or enhancement when merging that change you may need to adjust the version of the develop branch and the contribution to accommodate it.