Extension Repositories (aka Extension Registry) for Dynamically-loaded Extensions

1. Purpose

This page will capture design ideas for creating an Apache NiFi registry for extensions and templates, allowing them to be easily pulled into a NiFi instance, and thus reducing the amount of bundles distributed with an Apache NiFi release.

2. Problem

Today NiFi provides a mechanism to introduce new components (e.g., NAR) and share flow templates (pre-configured flows). For components (Processors, ControllerServices etc.) the current mechanism assumes packages (NARs) containing such components exist in some predefined location on local file system as they are typically distributed with NiFi distribution. For templates there is a UI-based import/export functionality giving user a little more control as to what is available to NiFi.

While fairly simple and well understood, current mechanism does expose certain challenges:

1. Given that NiFi comes distributed with all supported components its distribution size became quite large housing a series of extension sets that may or may not be of use by a given flow. As more components are introduced, the current mechanism will become unsustainable.
2. Regardless if there is an ***intent to use,*** every component is loaded into the JVM (classes are loaded to produce documentation and multiple instance are created due to the pre-existing bug **NIFI-1318** - Getting issue details... STATUS).
3. Management and sharing of artifacts (NARs, templates etc) is a manual process
    1. On top of that exported templates contain flow/component state (running/stopped), some identifiers are tied to the identifier scheme of the system on which the template was made which may present security risk.
4. ***No versioning*** support for components and/or templates
5. No ability to introduce new components once NiFi is started. Components must exist in NiFi prior to NiFi startup and loaded into NiFi (see #2).
6. Third-party vendors are locked out of participation due to legal constraints of including a component with non-ASF compatible licenses or including components that may depend on other components with non-ASF compatible licenses.
7. Repetitive library distribution. While each component is designed to perform a very specific task, most of them still depend on common libraries (e.g., log4j, spring, etc.). Such libraries are included as part of individual NARs creating a possibility to where several NARs in NiFi distribution contain the same library.

By providing a configurable and centralized ***extension registry*** similar to the one used by other products that expose *plug-and-play extension model* and thus heavily rely on community participation (Grails plug-ins), most (if not all) challenges described above could be addressed, facilitating even greater NiFi adoption.

3. Proposed Extension Registry Features

Below is the list of features that we may want to consider

1. Publish NAR/Template.
2. **Version**, presumably following a well known and recognized convention (e.g., Maven - artifactId:groupId:version).
    1. This will expose ability to have multiple versions of the same component available to the user.
    2. Additionally, NAR/Template may be tied to a specific version (or range of versions) of NiFi
3. Access components documentation regardless if such components are available locally.

4. Pull NAR/Template. This essentially implies bringing NAR/Template from the remote location to a running instance of NiFi. However, I don't believe this implies that user has to know about local/remote availability of the component. In fact in best case scenario user experience must not change. For example, user would have a familiar browser window for Processors as they do today, but such window would be populated with what is available to the user from the extension registry. If a particular component is already available locally then that is where it will be loaded from and if not then NiFi should transparently pull such component, essentially caching NAR /Templates locally (similar to local Maven cache).
5. Browse and/or search for NARs or templates within and outside of NiFi UI. This implies that the ***extension registry*** must have the ability to be accessed with the conventional web browser.
6. Configure location of extension registry. Similar to Maven, user's must be able to configure the location of the ***extension registry.***

4. Registry Examples

This section will describe how some other projects are tackling the same type of problem.

JFrog Bintray

Quoting their main page *"Bintray gives developers full control over how they store, publish, download, promote and distribute software with advanced features that fully automate the software distribution process"*

Some of the key features;

- Free for hosting OS artifacts
- Multiple repository/packaging styles including Maven, Docker RPM etc., essentially supporting versioning feature described in "*Proposed Features*"
- Browsing and Search capabilities described in "*Proposed Features*"
- Documentation features described in "*Proposed Features*"
- REST API to integrated with running NIFi instance to expose Browse, Search and provide access to documentation.
- Automated plug-ins (Maven, Gtradle etc) for automated publishing of the artifacts

Spark-Packages

- Requires authentication with a GitHub account.
- Registering a package requires pointing to a public GitHub repo owned by the given user, with a LICENSE and README.
- A name and description are also provided during registration.
- There is a command line tool to help start new packages and publish them.
- The web site lets a user search for packages and view details of each package.
- The details of a package show how to use the given package with various tools such as the spark-shell, sbt, and/or Maven, see example.
- For Maven it shows a dependency and repository snippet that could be added to a pom, and there seems to be a repository.
- Packages can also be voted on and tagged.

5. Use Cases

> NOTE: Hereafter, we use the term "extension" to mean all types of extension including both NAR packages and Templates, as well as future types not yet defined. (We need some generic term, and it didn't seem necessary to limit "extension" to just NAR packages.)

We are guided by the following use cases:

1. To keep NiFi small and lightweight, both in distribution and deployment, we want to separate the 40 current NARs from the core distro and supp ort dynamically loading them, both from a local filesystem repository and a remote web-based repository. Appropriate security support must be part of the implementation.

2. A User, using the NiFi GUI to construct a new Workflow, needs the ability to DISCOVER new Processors and other extensions they can use. T his suggests interactions that allow them to point NiFi at known-safe repositories, and then browse or otherwise discover the Extensions available therein (without loading all those extensions, some of which are quite large). This implies the need to obtain from the repo, manage, and present, METADATA and descriptive info about the extensions separately from the extensions themselves.

3. The community desires to share Templates. Templates are XML files with dependencies on Processors and other resources they may specify. The extension and repository model for NiFi should support sharing Templates and their dependencies in a natural and secure way.

4. One of us is looking at adding an Apache Camel Processor pair to NiFi (NIFI-1842). Camel supports connecting to over 200 end-point types no t yet supported natively in NiFi, and a Camel Processor could give us access to many of them with minimal additional work. Each of those end-points is already packaged as a dynamically-loadable extension to Camel. The packaging is as a JAR. We desire to enable NiFi to manage Camel extensions as NiFi plug-ins, without having to repackage each as a NAR, and without needing separate non-Camel repositories for them. In the future, other new NiFi extensions may also have need for custom sub-extensions of their own.

5. ~~We would like the set of extension types to be extensible, itself. Each extension type needs its own integration with core NiFi. Besides loading classes and other resources, those resources must be registered with different parts of the GUI and other components of NiFi. The extensibility mechanism should recognize and serve this need as well as possible.~~ To reduce the complexity of this project, we propose to NOT include extensibility of the set of extension types, nor repository types. Extending these will require core code changes, for now.

6. Design Proposal

A. Principal Abstractions

The two principal abstractions are **ExtensionSpec** and ~~**ExternalRepository**~~ **ExtensionRepository**.

An **ExtensionRepository** instance represents a repository serving extension packages, and provides APIs for NiFi to access such repositories in a standard way. It is recommended that there always be at least one trusted ExtensionRepository, designated the System Repository, installed as part of the NiFi deployment, and serving the set of standard NiFi extensions published as part of NiFi releases. ExtensionRepository implementations provide the following functionality:

- Present metadata about some or all of the extension packages in the repo, to facilitate human-interactive discovery.
- Manage information about the security signature of extension packages, and only deliver packages that are properly signed, by a signing authority recognized and approved at the system level.
- Manage information about the dependencies of the extension, which are declared in metadata, and only deliver packages whose dependencies are either:
    - already available in the system, or System Repository,
    - or are available from the *same* repository instance.
    - (For dependencies which are not available in the system nor in the same repository, delivery of the specified package will fail until the dependencies are resolved by User action. This avoids recursive calls to repository code, and issues of cross-access between repositories without human approval. Note extensions in the System Repository should not have outside dependencies; the System Repo should have closure.)
- Deliver individual extension package files, and where possible their dependencies, on demand.

ExtensionRepositories don't know anything about the internals of the extension package, or the packaging itself. Their responsibility ends after delivering, to an agreed location in the local filesystem, the concrete package file that can be loaded by some other mechanism appropriate to the extension type.

Our initial implementation will support as sub-classes:

- Filesystem directory-based repositories mounted locally on the NiFi server
- Maven repositories enabled in the NiFi server's maven configuration.

In this implementation, the set of supported repository types (ExtensionRepository sub-classes) can only be extended via core code changes. We choose not to make ExtensionRepository an extension type, to assure community review of highly security-sensitive code, and to control the scope of this project.

An **ExtensionSpec** instance is an envelope holding the metadata needed to support identification, discovery, delivery, and loading of a single extension package and its dependencies. This includes such metadata as extension name, extension type, packaging type, version, repository containing the extension, address or locator information within that repository, human-readable description, and dependency list. If more than a couple lines of descriptive text are required to assist human-interactive discovery, the metadata may include a pointer to a supplementary documentation file (see sub-section 'B' below). When an ExtensionRepository presents metadata about an extension, it constructs an ExtensionSpec instance. When the GUI presents extension info to the user prior to loading that extension, it obtains that info from an ExtensionSpec instance. The ExtensionSpec class is sub-classed at need, when different sets of metadata are needed for the purposes of different kinds of extension. However, many different kinds of extension may share a common ExtensionSpec subclass, being distinguished by the extension type and packaging type member fields. (In other words, don't expect a different ExtensionSpec subclass for every different kind of extension, as we found no need for this.)

Our initial implementation will support:

- All extension types already supported by the NAR sideloader, specifically including Processor, Controller Service, and Reporting Task extensions, all packaged as NARs
- Templates packaged as XMLs

In this implementation, the set of supported extension types and ExtensionSpec sub-classes can only be extended via core code changes. Furthermore, we choose to leave the extension loader code (load and registration process) for each extension type in core code, rather than try to incorporate this highly extension-type-specific code into this project. This is partly to reduce the complexity of the implementation by not having to refactor lots of existing loader code, and partly because we didn't see the need since adding new extension types will require core code changes anyway.

B. Metadata, Descriptive Documentation, and Security Signatures

For both the filesystem-based repository and the maven repository, we propose to require that each extension be distributed as a set of 6 files:

- a single-file package containing the extension itself, as a NAR or other file format appropriate to the extension type
- a maven POM file for the extension, containing certain mandatory XML elements, from which the extension's metadata and dependencies are obtained
- MD5 and SHA1 signature files for each of the package file and POM file.

These requirements mirror the requirements for a normal Maven repository deployment, and are not unreasonably burdensome for a filesystem-based repository deployment. Other repository types may share the same means of managing the metadata and security signature requirements, or they may choose to manage this info in other ways.

Some types of extension may need more than a brief textual description; for instance, a user exploring Templates would benefit from a rendered flow diagram for each Template. Such info is not appropriate in a POM file, so in addition to the above, we propose, optionally and only when needed:

- a documentation file (in adoc format), including the signature of the described extension
- and MD5 and SHA1 signature files for the adoc file

C. Note about Registration as part of Loading

To dynamically load an extension, it is not sufficient to simply ingest the code or content of the extension into the JVM. It is also necessary that the clients of the code or content become "aware" that the new material is available. This may be done in a variety of ways, depending on the structure of the client code, and may be referred to as registration, triggering, alerting, or re-scanning. Today's NAR loader supports registration lists, via the ExtensionMapping class, for processors, controller services, and reporting tasks. We haven't yet determined if the NAR loader provides for the registration needs of any other extension types. As noted above, we have declined to refactor existing loading code as part of the scope of this project, so loading and registration will remain an extension-type-specific part of core code.