NiFi Components

Overview

New contributors to NiFi often have the same question: Where can I start? When you look at a project like NiFi, one can imagine that there are quite a few moving parts, so understanding how all of these parts fit together can in itself become pretty daunting. This documentation is intended to be a very high level description of the components that make up NiFi and how those components fit together. It is not intended to go into great detail about any of these components. There are other documents that exist (and more to come) that delve a bit deeper into the design of some of these individual components. The goal of this document is to help developers who are new to NiFi come up to speed on some of terminology and understand how the different components that make up the platform interact with one another.

FlowFile

We will begin the discussion with the FlowFile. This is the abstraction that NiFi provides around a single piece of data. A FlowFile may represent structured data, such as a JSON or XML message, or may represent unstructured data, such as an image. A FlowFile is made up of two parts: content and attributes. The content is simply a stream of bytes, which can represent any type of data. The content itself is not stored within the FlowFile, however. Rather, the FlowFile simply stores a reference to the content, which lives within the Content Repository. Content is accessed only by means of the Process Session, which is capable of communicating with the Content Repository itself. The attributes are key-value pairs that are associated with the data. These attributes provide context along with the data, which allows the data to be efficiently routed and reasoned about without parsing the content.

Processor

This is the most commonly used component in NiFi and tends to be the easiest place for newcomers to jump in. A Processor is a component that is responsible for bringing data into the system, pushing data out to other systems, or performing some sort of enrichment, extraction, transformation, splitting, merging, or routing logic for a particular piece of data. Common Design Patterns for Processors are discussed in the Developer Guide.

The Processor is an extension point, and its API will not change from one minor release of NiFi to another but may change with a new major release of NiFi.

Processor Node

The Processor Node is essentially a wrapper around a Processor and maintains state about the Processor itself. The Processor Node is responsible for maintaining, among other things, state about a Processor's positioning on the graph, the configured properties and settings of the Processor, its scheduled state, and the annotations that are used to describe the Processor.

By abstracting these things away from the Processor itself, we are able to ensure that the Processor is unable to change things that it should not, such as the configured values for properties, as allowing a Processor to change this information can lead to confusion. Additionally, it allows us to simplify the code required to create a Processor, as this state information is automatically managed by the framework.

Reporting Task

A Reporting Task is a NiFi extension point that is capable of reporting and analyzing NiFi's internal metrics in order to provide the information to external resources or report status information as bulletins that appear directly in the NiFi User Interface. Unlike a Processor, a Reporting Task does not have access to individual FlowFiles. Rather, a Reporting Task has access to all Provenance Events, bulletins, and the metrics shown for components on the graph, such as FlowFiles In, Bytes Read, and Bytes Written.

The Reporting Task is an extension point, and its API will not change from one minor release of NiFi to another but may change with a new major release of NiFi.

Controller Service

The Controller Service is a mechanism that allows state or resources to be shared across multiple components in the flow. The SSLContextService, for instance, allows a user to configure SSL information only once and then configure any number of resources to use that configuration. Other Controller Services are used to share configuration. For example, if a very large dataset needs to be loaded, it will generally make sense to use a Controller Service to load the dataset. This allows multiple Processors to make use of this dataset without having to load the dataset multiple times.

The Controller Service is an extension point, and its API will not change from one minor release of NiFi to another but may change with a new major release of NiFi.

Process Session

The Process Session (often referred to simply as a "session") provides Processors access to FlowFiles and provides transactional behavior across the tasks that are performed by a Processor. The session provides `get()` methods for obtaining access to FlowFiles that are queued up for a Processor, methods to read from and write to the contents of a FlowFile, add and remove FlowFiles from the flow, add and remove attributes from a FlowFile, and route a FlowFile to a particular relationship. Additionally, the session provides access to the ProvenanceReporter that is used by Processors to emit Provenance Events.

Once a Processor is finished performing its task, the Processor has the ability to either commit or rollback the session. If a Processor rolls back the session, the FlowFiles that were accessed during that session will all be reverted to their previous states. Any FlowFile that was added to the flow will be destroyed. Any FlowFile that was removed from the flow will be re-queued in the same queue that it was pulled from. Any FlowFile that was modified will have both its contents and attributes reverted to their previous values, and the FlowFiles will all be re-queued into the FlowFile Queue that they were pulled from. Additionally, any Provenance Events will be discarded.

If a Processor instead chooses to commit the session, the session is responsible for updating the FlowFile Repository and Provenance Repository with the relevant information. The session will then add the FlowFiles to the Processor's outbound queues (cloning as necessary, if the FlowFile was transferred to a relationship for which multiple connections have been established).

Process Context

The Process Context provides a bridge between a Processor and its associated Processor Node. It provides information about about the Processor's current configuration, as well as the ability to "yield," or signal to the framework that it is unable to perform any work for a short period of time so the framework should not not waste resources scheduling the Processor to run. The Process Context also provides mechanisms for accessing the Controller Services that are available, so that Processors are able to take advantage of shared logic or shared resources.

FlowFile Repository

The FlowFile Repository is responsible for storing the FlowFiles' attributes and state, such as creation time and which FlowFile Queue the FlowFile belongs in. The default implementation is the WriteAheadFlowFileRepository, which persists the information to a write-ahead log that is periodically "checkpointed". This allows extremely high transaction rates, as the files that it writes to are "append-only," so the OutputStreams are able to be kept open. Periodically, the repository will checkpoint, meaning that it will begin writing to new write-ahead logs, write out the state of all FlowFiles at that point in time, and delete the old write-ahead logs. This prevents the write-ahead logs from growing indefinitely. For more information on the design and implementation of this repository, see the NiFi Write-Ahead Log Implementation page.

Note: While the FlowFile Repository is pluggable, it is considered a 'private API' and its interface could potentially be changed between minor versions of NiFi. It is, therefore, not recommended that implementations be developed outside of the NiFi codebase.

Content Repository

The Content Repository is responsible for storing the content of FlowFiles and providing mechanisms for reading the contents of a FlowFile. This abstraction allows the contents of FlowFiles to be stored independently and efficiently based on the underlying storage mechanism. The default implementation is the FileSystemRepository, which persists all data to the underlying file system.

Note: While the Content Repository is pluggable, it is considered a 'private API' and its interface could potentially be changed between minor versions of NiFi. It is, therefore, not recommended that implementations be developed outside of the NiFi codebase.

Provenance Repository

The Provenance Repository is responsible for storing, retrieving, and querying all Data Provenance Events. Each time that a FlowFile is received, routed, cloned, forked, modified, sent, or dropped, a Provenance Event is generated that details this information. The event contains information about what the Event Type was, which FlowFile(s) were involved, the FlowFile's attributes at the time of the event, details about the event, and a pointer to the Content of the FlowFile before and after the event occurred (which allows a user to understand how that particular event modified the FlowFile).

The Provenance Repository allows this information to be stored about each FlowFile as it traverses through the system and provides a mechanism for assembling a "Lineage view" of a FlowFile, so that a graphical representation can be shown of exactly how the FlowFile was handled. In order to determine which lineages to view, the repository exposes a mechanism whereby a user is able to search the events and associated FlowFile attributes.

The default implementation is PersistentProvenanceRepository. This repository stores all data immediately to disk-backed write-ahead log and periodically "rolls over" the data, indexing and compressing the data. The search capabilities are provided by an embedded Lucene engine. For more information on how this repository is designed and implemented, see the Persistent Provenance Repository Design page.

Note: While the Provenance Repository is pluggable, it is considered a 'private API' and its interface could potentially be changed between minor versions of NiFi. It is, therefore, not recommended that implementations be developed outside of the NiFi codebase.

Process Scheduler

In order for a Processor or a Reporting Task to be invoked, it needs to be scheduled to do so. This responsibility belongs to the Process Scheduler. In addition to scheduling Processors and Reporting Task, the scheduler is also responsible for scheduling framework tasks to run at periodic intervals and maintaining the schedule state of each component, as well as the current number of active threads. The Process Scheduler is able to inspect a particular component in order to determine which Scheduling Strategy to use (Cron Driven, Timer Driven, or Event Driven), as well as the scheduling frequency.

FlowFile Queue

Though it sounds sufficiently simple, the FlowFile Queue is responsible for implementing quite a bit of logic. In addition to queuing the FlowFiles for another component to pull from, the FlowFile Queue must also be able to prioritize the data following the user's prioritization rules. The queue keeps state about the number of FlowFiles as well as the data size of those FlowFiles and must keep state about the number of "in-flight" FlowFiles - those that have been pulled from the queue but have not yet been removed from the system or transferred to another queue.

When an instance of NiFi has a very large number of active FlowFiles, the attributes associated with those FlowFiles can be quite a burden on the JVM's heap. To alleviate this problem, the framework may choose to "swap out" some of the FlowFiles, writing the attributes to disk and removing them from the JVM's heap when a queue grows very large and later "swap in" those FlowFiles. During this process, the FlowFile Queue is also responsible for keeping track of the number of FlowFiles and size of the FlowFiles' contents so that accurate numbers can be reported to users.

Finally, the FlowFile Queue is also responsible for maintaining state about backpressure and FlowFile Expiration. Backpressure is the mechanism by which a user is able to configure the flow to temporarily stop scheduling a given component to run when its output queue is too full. By doing this, we are able to cause the flow to stop accepting incoming data for a short period, or to route data differently. This provides us the ability to prevent resource exhaustion. In a clustered environment, this also allows a particular node that is falling for one reason or another to avoid ingesting data so that other nodes in the cluster that are more capable can handle the workload.

FlowFile Expiration is the mechanism by which data is eventually purged from the system because it is no longer of value. It can be thought of as a flow's pressure release valve. This is used, for instance, in an environment when there is not enough bandwidth to send all of the data to its desired destination. In such a case, the node will eventually run out of resources. In order to avoid this, the user can configure a queue to expire data that reaches a certain age. For example, the user can indicate that data that is one hour old should automatically be purged. This capability is then coupled with the ability to prioritize the data so that the most important data is always sent first and the less important data eventually expires.

FlowFile Prioritizer

A core tenant of NiFi is that of data prioritization. The user should have the ability to prioritize the data in whatever order makes sense for a particular dataflow at a particular point in time. This is especially important for time-sensitive data and becomes even more important when processing data in an environment where the rate of data acquisition exceeds that rate at which the data can be egressed. In such an environment, it is important to be able to egress the data in such a way that the most important data is sent first, leaving the less important data to be sent when the bandwidth allows for it, or eventually be aged off.

This tenant is realized through the user of FlowFile Prioritizers. The FlowFile Queue is responsible for ensuring the data is ordered in the way that the user has chosen. This is accomplished by applying FlowFile Prioritizers to the queue. A FlowFile Prioritizer has access to all FlowFile information but does not have access to the data that it points to. The Prioritizer can then compare two FlowFiles in order to determine which should be made available first.

The Prioritizer is an extension point, and its API will not change from one minor release of NiFi to another but may change with a new major release of NiFi.

Flow Controller

In order for NiFi's User Interface to display the wealth of information that it renders, it must have some place to gather that information. The Flow Controller can be thought of as the bridge between the Web Tier and the back end components of a NiFi node. It is largely responsible for the initialization of the flow, the creation of components in the flow, and the coordination between those components. In a more general sense, the Flow Controller is responsible for maintaining system-wide state. This includes the state of the flow itself, and is responsible for coordinating connection and participation in a NiFi cluster.

Cluster Manager

Whereas the Flow Controller is responsible for maintaining system-wide state about a particular node, the Cluster Manager is responsible for maintaining system-wide state about an entire cluster. This includes information such as which nodes are in the cluster and the states of those nodes. Additionally, the Cluster Manager is responsible for acting as the bridge between the Web Tier and the cluster state. This includes replicating requests from the user to all nodes in the cluster and merging their responses.

An instance of NiFi will create either a Cluster Manager or a Flow Controller but never both. Which component is created is driven by the configuration found in the `nifi.properties` file.

Authority Provider

When NiFi is configured, it can be configured to run in secure mode, using SSL to access the web endpoints or to run in non-secure mode, where all endpoints are access anonymously. If running in secure mode, the Authority Provider is responsible for determining which users have access to perform which operations. It accomplishes this by providing a list of Authorities, or Roles, that a given user is allowed to possess. Note, it does not indicate which Roles a user actually has, only which Roles a user is allowed to have. A NiFi user who has the Role `ADMIN` is then able to grant each of those Roles to the given user. In this way, a central authority service can be used to dictate which Roles a given user is allowed to have, but just because a user is allowed to be granted the `ADMIN` role, for example, does not mean that the user should have the `ADMIN` role for a particular instance.

Each endpoint that is defined must also define which Roles are authorized to access the endpoint. A user is then able to access an endpoint if and only if the user has one of the specified Roles.

The Authority Provider is an extension point, and its API will not change from one minor release of NiFi to another but may change with a new major release of NiFi.

*-Resources

NiFi's User Interface shows only information that is available via the NiFi RESTful API. This is accomplished by accessing the different endpoints that are defined in the *-Resource classes. For example, the ProcessorResource class is responsible for defining the endpoints that are used to interact with the different Processors in the flow, including the addition and removal of Processors. The ProvenanceResource class is responsible for defining the endpoints that are used to interact with the Provenance Repository, such as executing queries.

All of the Resource components are found in the `nifi-web-api` module, in the `org.apache.nifi.web.api` package.

Bootstrap

In order for an organization to be able to depend on NiFi to handle their dataflows in an automated fashion, the organization needs to be able to rely on NiFi to handle unexpected conditions and to be able to startup and shutdown in an easy-to-use, consistent manner. The Bootstrap module is responsible for achieving these goals. The Bootstrap allows the user to easily configure how the NiFi JVM will be started by relying on the configuration provided in the `bootstrap.properties` file. This allows the application to easily be started with Remote Debugging enabled, with specified minimum and maximum heap sizes, and with other important JVM options.

Once the NIFi application has been started, the bootstrap is then responsible for monitoring NiFi and restarting the service as needed in order to ensure that the application continues to provide reliable dataflow.

NarClassLoader

In a containerized environment like NiFi, it is important to allow different extension points to have arbitrary dependencies without those dependencies affecting other, unrelated extension points. In Java, the mechanism for doing this is the ClassLoader. NiFi defines its own implementation of the ClassLoader, the NarClassLoader. Each NiFi Archive (NAR) has its own NarClassLoader that is responsible for loading the classes defined in that NAR. Additionally, it is responsible for providing native library isolation, so that a Processor, for example, can depend on native libraries that are not shared with other components. This is accomplished by placing those native libraries in the NAR's `native` directory.

For more information about NiFi Archives and how they are used, see the NiFi Archives (NARs) section in the Developer Guide.