Nifi Development

NiFi Components

NiFi provides several extension points to provide developers the ability to add functionality to the application to meet their needs. The following list provides a high-level description of the most common extension points:

* Processor
  + The Processor interface is the mechanism through which NiFi exposes access to [FlowFile](https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#flowfile)s, their attributes, and their content. The Processor is the basic building block used to comprise a NiFi dataflow. This interface is used to accomplish all of the following tasks:
    - Create FlowFiles
    - Read FlowFile content
    - Write FlowFile content
    - Read FlowFile attributes
    - Update FlowFile attributes
    - Ingest data
    - Egress data
    - Route data
    - Extract data
    - Modify data
* ReportingTask
  + The ReportingTask interface is a mechanism that NiFi exposes to allow metrics, monitoring information, and internal NiFi state to be published to external endpoints, such as log files, e-mail, and remote web services.
* ControllerService
  + A ControllerService provides shared state and functionality across Processors, other ControllerServices, and ReportingTasks within a single JVM. An example use case may include loading a very large dataset into memory. By performing this work in a ControllerService, the data can be loaded once and be exposed to all Processors via this service, rather than requiring many different Processors to load the dataset themselves.
* FlowFilePrioritizer
  + The FlowFilePrioritizer interface provides a mechanism by which [FlowFile](https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#flowfile)s in a queue can be prioritized, or sorted, so that the FlowFiles can be processed in an order that is most effective for a particular use case.
* AuthorityProvider
  + An AuthorityProvider is responsible for determining which privileges and roles, if any, a given user should be granted.

Processor API

The Processor is the most widely used Component available in NiFi. Processors are the only Component to which access is given to create, remove, modify, or inspect FlowFiles (data and attributes).

All Processors are loaded and instantiated using Java’s ServiceLoader mechanism. This means that all Processors must adhere to the following rules:

* The Processor must have a default constructor.
* The Processor’s JAR file must contain an entry in the META-INF/services directory named org.apache.nifi.processor.Processor. This is a text file where each line contains the fully-qualified class name of a Processor.

While Processor is an interface that can be implemented directly, it will be extremely rare to do so, as the org.apache.nifi.processor.AbstractProcessor is the base class for almost all Processor implementations. The AbstractProcessor class provides a significant amount of functionality, which makes the task of developing a Processor much easier and more convenient. For the scope of this document, we will focus primarily on the AbstractProcessor class when dealing with the Processor API.

*Concurrency Note*

NiFi is a highly concurrent framework. This means that all extensions must be thread-safe. If unfamiliar with writing concurrent software in Java, it is highly recommended that you familiarize yourself with the principles of Java concurrency.

Supporting API

In order to understand the Processor API, we must first understand - at least at a high level - several supporting classes and interfaces, which are discussed below.

FlowFile

A FlowFile is a logical notion that correlates a piece of data with a set of Attributes about that data. Such attributes include a FlowFile’s unique identifier, as well as its name, size, and any number of other flow-specific values. While the contents and attributes of a FlowFile can change, the FlowFile object is immutable. Modifications to a FlowFile are made possible by the ProcessSession.

The core attributes for FlowFiles are defined in the org.apache.nifi.flowfile.attributes.CoreAttributes enum. The most common attributes you’ll see are filename, path and uuid. The string in quotes is the value of the attribute within the CoreAttributes enum.

* Filename ("filename"): The filename of the FlowFile. The filename should not contain any directory structure.
* UUID ("uuid"): A Universally Unique Identifier assigned to this FlowFile that distinguishes the FlowFile from other FlowFiles in the system.
* Path ("path"): The FlowFile’s path indicates the relative directory to which a FlowFile belongs and does not contain the filename.
* Absolute Path ("absolute.path"): The FlowFile’s absolute path indicates the absolute directory to which a FlowFile belongs and does not contain the filename.
* Priority ("priority"): A numeric value indicating the FlowFile priority.
* MIME Type ("mime.type"): The MIME Type of this FlowFile.
* Discard Reason ("discard.reason"): Specifies the reason that a FlowFile is being discarded.
* Alternative Identifier ("alternate.identifier"): Indicates an identifier other than the FlowFile’s UUID that is known to refer to this FlowFile.

ProcessSession

The ProcessSession, often referred to as simply a "session," provides a mechanism by which FlowFiles can be created, destroyed, examined, cloned, and transferred to other Processors. Additionally, a ProcessSession provides mechanism for creating modified versions of FlowFiles, by adding or removing attributes, or by modifying the FlowFile’s content. The ProcessSession also exposes a mechanism for emitting [Provenance Events](https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#provenance_events) that provide for the ability to track the lineage and history of a FlowFile. After operations are performed on one or more FlowFiles, a ProcessSession can be either committed or rolled back.

ProcessContext

The ProcessContext provides a bridge between a Processor and the framework. It provides information about how the Processor is currently configured and allows the Processor to perform Framework-specific tasks, such as yielding its resources so that the framework will schedule other Processors to run without consuming resources unnecessarily.

PropertyDescriptor

PropertyDescriptor defines a property that is to be used by a Processor, ReportingTask, or ControllerService. The definition of a property includes its name, a description of the property, an optional default value, validation logic, and an indicator as to whether or not the property is required in order for the Processor to be valid. PropertyDescriptors are created by instantiating an instance of the PropertyDescriptor.Builder class, calling the appropriate methods to fill in the details about the property, and finally calling the build method.

Validator

A PropertyDescriptor MUST specify one or more Validators that can be used to ensure that the user-entered value for a property is valid. If a Validator indicates that a property value is invalid, the Component will not be able to be run or used until the property becomes valid. If a Validator is not specified, the Component will be assumed invalid and NiFi will report that the property is not supported.

ValidationContext

When validating property values, a ValidationContext can be used to obtain ControllerServices, create PropertyValue objects, and compile and evaluate property values using the Expression Language.

PropertyValue

All property values returned to a Processor are returned in the form of a PropertyValue object. This object has convenience methods for converting the value from a String to other forms, such as numbers and time periods, as well as providing an API for evaluating the Expression Language.

Relationship

Relationships define the routes to which a FlowFile may be transferred from a Processor. Relationships are created by instantiating an instance of the Relationship.Builder class, calling the appropriate methods to fill in the details of the Relationship, and finally calling the build method.

StateManager

The StateManager provides Processors, Reporting Tasks, and Controller Services a mechanism for easily storing and retrieving state. The API is similar to that of ConcurrentHashMap but requires a Scope for each operation. The Scope indicates whether the state is to be retrieved/stored locally or in a cluster-wide manner. For more information, see the [State Manager](https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#state_manager) section.

ProcessorInitializationContext

After a Processor is created, its initialize method will be called with an InitializationContext object. This object exposes configuration to the Processor that will not change throughout the life of the Processor, such as the unique identifier of the Processor.

ComponentLog

Processors are encouraged to perform their logging via the ComponentLog interface, rather than obtaining a direct instance of a third-party logger. This is because logging via the ComponentLog allows the framework to render log messages that exceeds a configurable severity level to the User Interface, allowing those who monitor the dataflow to be notified when important events occur. Additionally, it provides a consistent logging format for all Processors by logging stack traces when in DEBUG mode and providing the Processor’s unique identifier in log messages.

AbstractProcessor API

Since the vast majority of Processors will be created by extending the AbstractProcessor, it is the abstract class that we will examine in this section. The AbstractProcessor provides several methods that will be of interest to Processor developers.

Processor Initialization

When a Processor is created, before any other methods are invoked, the init method of the AbstractProcessor will be invoked. The method takes a single argument, which is of type ProcessorInitializationContext. The context object supplies the Processor with a ComponentLog, the Processor’s unique identifier, and a ControllerServiceLookup that can be used to interact with the configured ControllerServices. Each of these objects is stored by the AbstractProcessor and may be obtained by subclasses via the getLogger, getIdentifier, and getControllerServiceLookup methods, respectively.

Exposing Processor’s Relationships

In order for a Processor to transfer a FlowFile to a new destination for follow-on processing, the Processor must first be able to expose to the Framework all of the Relationships that it currently supports. This allows users of the application to connect Processors to one another by creating Connections between Processors and assigning the appropriate Relationships to those Connections.

A Processor exposes the valid set of Relationships by overriding the getRelationships method. This method takes no arguments and returns a Set of Relationship objects. For most Processors, this Set will be static, but other Processors will generate the Set dynamically, based on user configuration. For those Processors for which the Set is static, it is advisable to create an immutable Set in the Processor’s constructor or init method and return that value, rather than dynamically generating the Set. This pattern lends itself to cleaner code and better performance.

Exposing Processor Properties

Most Processors will require some amount of user configuration before they are able to be used. The properties that a Processor supports are exposed to the Framework via the getSupportedPropertyDescriptors method. This method takes no arguments and returns a List of PropertyDescriptor objects. The order of the objects in the List is important in that it dictates the order in which the properties will be rendered in the User Interface.

A PropertyDescriptor object is constructed by creating a new instance of the PropertyDescriptor.Builder object, calling the appropriate methods on the builder, and finally calling the build method.

While this method covers most of the use cases, it is sometimes desirable to allow users to configure additional properties whose name are not known. This can be achieved by overriding the getSupportedDynamicPropertyDescriptor method. This method takes a String as its only argument, which indicates the name of the property. The method returns a PropertyDescriptor object that can be used to validate both the name of the property, as well as the value. Any PropertyDescriptor that is returned from this method should be built setting the value of isDynamic to true in the PropertyDescriptor.Builder class. The default behavior of AbstractProcessor is to not allow any dynamically created properties.

Validating Processor Properties

A Processor is not able to be started if its configuration is not valid. Validation of a Processor property can be achieved by setting a Validator on a PropertyDescriptor or by restricting the allowable values for a property via the PropertyDescriptor.Builder’s allowableValues method or identifiesControllerService method.

There are times, though, when validating a Processor’s properties individually is not sufficient. For this purpose, the AbstractProcessor exposes a customValidate method. The method takes a single argument of type ValidationContext. The return value of this method is a Collection of ValidationResult objects that describe any problems that were found during validation. Only those ValidationResult objects whose isValid method returns false should be returned. This method will be invoked only if all properties are valid according to their associated Validators and Allowable Values. I.e., this method will be called only if all properties are valid in-and-of themselves, and this method allows for validation of a Processor’s configuration as a whole.

Responding to Changes in Configuration

It is sometimes desirable to have a Processor eagerly react when its properties are changed. The onPropertyModified method allows a Processor to do just that. When a user changes the property values for a Processor, the onPropertyModified method will be called for each modified property. The method takes three arguments: the PropertyDescriptor that indicates which property was modified, the old value, and the new value. If the property had no previous value, the second argument will be null. If the property was removed, the third argument will be null. It is important to note that this method will be called regardless of whether or not the values are valid. This method will be called only when a value is actually modified, rather than being called when a user updates a Processor without changing its value. At the point that this method is invoked, it is guaranteed that the thread invoking this method is the only thread currently executing code in the Processor, unless the Processor itself creates its own threads.

Performing the Work

When a Processor has work to do, it is scheduled to do so by having its onTrigger method called by the framework. The method takes two arguments: a ProcessContext and a ProcessSession. The first step in the onTrigger method is often to obtain a FlowFile on which the work is to be performed by calling one of the get methods on the ProcessSession. For Processors that ingest data into NiFi from external sources, this step is skipped. The Processor is then free to examine FlowFile attributes; add, remove, or modify attributes; read or modify FlowFile content; and transfer FlowFiles to the appropriate Relationships.

When Processors are Triggered

A Processor’s onTrigger method will be called only when it is scheduled to run and when work exists for the Processor. Work is said to exist for a Processor if any of the following conditions is met:

* A Connection whose destination is the Processor has at least one FlowFile in its queue
* The Processors has no incoming Connections
* The Processor is annotated with the @TriggerWhenEmpty annotation

Several factors exist that will contribute to when a Processor’s onTrigger method is invoked. First, the Processor will not be triggered unless a user has configured the Processor to run. If a Processor is scheduled to run, the Framework periodically (the period is configured by users in the User Interface) checks if there is work for the Processor to do, as described above. If so, the Framework will check downstream destinations of the Processor. If any of the Processor’s outbound Connections is full, by default, the Processor will not be scheduled to run.

However, the @TriggerWhenAnyDestinationAvailable annotation may be added to the Processor’s class. In this case, the requirement is changed so that only one downstream destination must be "available" (a destination is considered "available" if the Connection’s queue is not full), rather than requiring that all downstream destinations be available.

Also related to Processor scheduling is the @TriggerSerially annotation. Processors that use this Annotation will never have more than one thread running the onTrigger method simultaneously. It is crucial to note, though, that the thread executing the code may change from invocation to invocation. Therefore, care must still be taken to ensure that the Processor is thread-safe!

### Restricted

A Restricted component is one that can be used to execute arbitrary unsanitized code provided by the operator through the NiFi REST API/UI or can be used to obtain or alter data on the NiFi host system using the NiFi OS credentials. These components could be used by an otherwise authorized NiFi user to go beyond the intended use of the application, escalate privilege, or could expose data about the internals of the NiFi process or the host system. All of these capabilities should be considered privileged, and admins should be aware of these capabilities and explicitly enable them for a subset of trusted users.

A Processor, Controller Service, or Reporting Task can be marked with the @Restricted annotation. This will result in the component being treated as restricted and will require a user to be explicitly added to the list of users who can access restricted components. Once a user is permitted to access restricted components, they will be allowed to create and modify those components assuming all other permissions are permitted. Without access to restricted components, a user will still be aware these types of components exist but will be unable to create or modify them even with otherwise sufficient permissions.

### State Manager

From the ProcessContext, ReportingContext, and ControllerServiceInitializationContext, components are able to call the getStateManager() method. This State Manager is responsible for providing a simple API for storing and retrieving state. This mechanism is intended to provide developers with the ability to very easily store a set of key/value pairs, retrieve those values, and update them atomically. The state can be stored local to the node or across all nodes in a cluster. It is important to note, however, that this mechanism is intended only to provide a mechanism for storing very 'simple' state. As such, the API simply allows a Map<String, String> to be stored and retrieved and for the entire Map to be atomically replaced. Moreover, the only implementation that is currently supported for storing cluster-wide state is backed by ZooKeeper. As such, the entire State Map must be less than 1 MB in size, after being serialized. Attempting to store more than this will result in an Exception being thrown. If the interactions required by the Processor for managing state are more complex than this (e.g., large amounts of data must be stored and retrieved, or individual keys must be stored and fetched individually) than a different mechanism should be used (e.g., communicating with an external database).

#### Scope

When communicating with the State Manager, all method calls require that a Scope be provided. This Scope will either be Scope.LOCAL or Scope.CLUSTER. If NiFi is run in a cluster, this Scope provides important information to the framework about how the operation should occur.

If state as stored using Scope.CLUSTER, then all nodes in the cluster will be communicating with the same state storage mechanism. If state is stored and retrieved using Scope.LOCAL, then each node will see a different representation of the state.

It is also worth noting that if NiFi is configured to run as a standalone instance, rather than running in a cluster, a scope of Scope.LOCAL is always used. This is done in order to allow the developer of a NiFi component to write the code in one consistent way, without worrying about whether or not the NiFi instance is clustered. The developer should instead assume that the instance is clustered and write the code accordingly.

#### Storing and Retrieving State

State is stored using the StateManager’s getState, setState, replace, and clear methods. All of these methods require that a Scope be provided. It should be noted that the state that is stored with the Local scope is entirely different than state stored with a Cluster scope. If a Processor stores a value with the key of My Key using the Scope.CLUSTER scope, and then attempts to retrieve the value using the Scope.LOCAL scope, the value retrieved will be null (unless a value was also stored with the same key using the Scope.CLUSTER scope). Each Processor’s state is stored in isolation from other Processors' state.

It follows, then, that two Processors cannot share the same state. There are, however, some circumstances in which it is very necessary to share state between two Processors of different types, or two Processors of the same type. This can be accomplished by using a Controller Service. By storing and retrieving state from a Controller Service, multiple Processors can use the same Controller Service and the state can be exposed via the Controller Service’s API.

#### Unit Tests

NiFi’s Mock Framework provides an extensive collection of tools to perform unit testing of Processors. Processor unit tests typically begin with the TestRunner class. As a result, the TestRunner class contains a getStateManager method of its own. The StateManager that is returned, however, is of a specific type: MockStateManager. This implementation provides several methods in addition to those defined by the StateManager interface, that help developers to more easily develop unit tests.

First, the MockStateManager implements the StateManager interface, so all of the state can be examined from within a unit test. Additionally, the MockStateManager exposes a handful of assert\* methods to perform assertions that the State is set as expected. The MockStateManager also provides the ability to indicate that the unit test should immediately fail if state is updated for a particular Scope.

### Reporting Processor Activity

Processors are responsible for reporting their activity so that users are able to understand what happens to their data. Processors should log events via the ComponentLog, which is accessible via the InitializationContext or by calling the getLogger method of AbstractProcessor.

Additionally, Processors should use the ProvenanceReporter interface, obtained via the ProcessSession’s getProvenanceReporter method. The ProvenanceReporter should be used to indicate any time that content is received from an external source or sent to an external location. The ProvenanceReporter also has methods for reporting when a FlowFile is cloned, forked, or modified, and when multiple FlowFiles are merged into a single FlowFile as well as associating a FlowFile with some other identifier. However, these functions are less critical to report, as the framework is able to detect these things and emit appropriate events on the Processor’s behalf. Yet, it is a best practice for the Processor developer to emit these events, as it becomes explicit in the code that these events are being emitted, and the developer is able to provide additional details to the events, such as the amount of time that the action took or pertinent information about the action that was taken. If the Processor emits an event, the framework will not emit a duplicate event. Instead, it always assumes that the Processor developer knows what is happening in the context of the Processor better than the framework does. The framework may, however, emit a different event. For example, if a Processor modifies both the content of a FlowFile and its attributes and then emits only an ATTRIBUTES\_MODIFIED event, the framework will emit a CONTENT\_MODIFIED event. The framework will not emit an ATTRIBUTES\_MODIFIED event if any other event is emitted for that FlowFile (either by the Processor or the framework). This is due to the fact that all [Provenance Events](https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#provenance_events) know about the attributes of the FlowFile before the event occurred as well as those attributes that occurred as a result of the processing of that FlowFile, and as a result the ATTRIBUTES\_MODIFIED is generally considered redundant and would result in a rendering of the FlowFile lineage being very verbose. It is, however, acceptable for a Processor to emit this event along with others, if the event is considered pertinent from the perspective of the Processor.