Apache NiFi 3

NiFi Developer's Guide

Date of Publish: 2019-03-15

https://docs.hortonworks.com/

Contents

Introduction

The intent of this Developer Guide is to provide the reader with the information needed to understand how Apache NiFi extensions are developed and help to explain the thought process behind developing the components. It provides an introduction to and explanation of the API that is used to develop extensions. It does not, however, go into great detail about each of the methods in the API, as this guide is intended to supplement the JavaDocs of the API rather than replace them. This guide also assumes that the reader is familiar with Java 7 and Apache Maven.

This guide is written by developers for developers. It is expected that before reading this guide, you have a basic understanding of NiFi and the concepts of dataflow.

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 FlowFiles, 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 FlowFiles 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 AuthorityProvide 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.

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 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 transfered 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.

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!

Component Lifecycle

The NiFi API provides lifecycle support through use of Java Annotations. The org.apache.nifi.annotations.lifecycle package contains several annotations for lifecycle management. The following Annotations may be applied to Java methods in a NiFi component to indicate to the framework when the methods should be called. For the discussion of Component Lifecycle, we will define a NiFi component as a Processor, ControllerServices, or ReportingTask.

@OnAdded

The @OnAdded annotation causes a method to be invoked as soon as a component is created. The component's initialize method (or init method, if subclasses AbstractProcessor) will be invoked after the component is constructed, followed by methods that are annotated with @OnAdded. If any method annotated with @OnAdded throws an Exception, an error will be returned to the user, and that component will not be added to the flow. Furthermore, other methods with this Annotation will not be invoked. This method will be called only once for the lifetime of a component. Methods with this Annotation must take zero arguments.

@OnEnabled

The @OnEnabled annotation can be used to indicate a method should be called whenever the Controller Service is enabled. Any method that has this annotation will be called every time a user enables the service. Additionally, each time that NiFi is restarted, if NiFi is configured to "auto-resume state" and the service is enabled, the method will be invoked.

If a method with this annotation throws a Throwable, a log message and bulletin will be issued for the component. In this event, the service will remain in an 'ENABLING' state and will not be usable. All methods with this annotation will then be called again after a delay. The service will not be made available for use until all methods with this annotation have returned without throwing anything.

Methods using this annotation must take either 0 arguments or a single argument of type org.apache.nifi.controller.ConfigurationContext.

Note that this annotation will be ignored if applied to a ReportingTask or Processor. For a Controller Service, enabling and disabling are considered lifecycle events, as the action makes them usable or unusable by other components. However, for a Processor and a Reporting Task, these are not lifecycle events but rather a mechanism to allow a component to be excluded when starting or stopping a group of components.

@OnRemoved

The @OnRemoved annotation causes a method to be invoked before a component is removed from the flow. This allows resources to be cleaned up before removing a component. Methods with this annotation must take zero arguments. If a method with this annotation throws an Exception, the component will still be removed.

@OnScheduled

This annotation indicates that a method should be called every time the component is scheduled to run. Because ControllerServices are not scheduled, using this annotation on a ControllerService does not make sense and will not be honored. It should be used only for Processors and Reporting Tasks. If any method with this annotation throws an Exception, other methods with this annotation will not be invoked, and a notification will be presented to the user. In this case, methods annotated with @OnUnscheduled are then triggered, followed by methods with the @OnStopped annotation (during this state, if any of these methods throws an Exception, those Exceptions are ignored). The component will then yield its execution for some period of time, referred to as the "Administrative Yield Duration," which is a value that is configured in the nifi.properties file. Finally, the process will start again, until all of the methods annotated with @OnScheduled have returned without throwing any Exception. Methods with this annotation may take zero arguments or may take a single argument. If the single argument variation is used, the argument must be of type ProcessContext if the component is a Processor or ConfigurationContext if the component is a ReportingTask.

@OnUnscheduled

Methods with this annotation will be called whenever a Processor or ReportingTask is no longer scheduled to run. At that time, many threads may still be active in the Processor's onTrigger method. If such a method throws an Exception, a log message will be generated, and the Exception will be otherwise ignored and other methods with this annotation will still be invoked. Methods with this annotation may take zero arguments or may take a single argument. If the single argument variation is used, the argument must be of type ProcessContext if the component is a Processor or ConfigurationContext if the component is a ReportingTask.

@OnStopped

Methods with this annotation will be called when a Processor or ReportingTask is no longer scheduled to run and all threads have returned from the onTrigger method. If such a method throws an Exception, a log message will be generated, and the Exception will otherwise be ignored; other methods with this annotation will still be invoked. Methods with this annotation are permitted to take either 0 or 1 argument. If an argument is used, it must be of type ConfigurationContext if the component is a ReportingTask or of type ProcessContext if the component is a Processor.

@OnShutdown

Any method that is annotated with the @OnShutdown annotation will be called when NiFi is successfully shut down. If such a method throws an Exception, a log message will be generated, and the Exception will be otherwise ignored and other methods with this annotation will still be invoked. Methods with this annotation must take zero arguments. Note: while NiFi will attempt to invoke methods with this annotation on all components that use it, this is not always possible. For example, the process may be killed unexpectedly, in which case it does not have a chance to invoke these methods. Therefore, while methods using this annotation can be used to clean up resources, for instance, they should not be relied upon for critical business logic.

Component Notification

The NiFi API provides notification support through use of Java Annotations. The org.apache.nifi.annotations.notification package contains several annotations for notification management. The following annotations may be applied to Java methods in a NiFi component to indicate to the framework when the methods should be called. For the discussion of Component Notification, we will define a NiFi component as a Processor, Controller Service, or Reporting Task.

@OnPrimaryNodeStateChange

The @OnPrimaryNodeStateChange annotation causes a method to be invoked as soon as the state of the Primary Node in a cluster has changed. Methods with this annotation should take either no arguments or one argument of type PrimaryNodeState. The PrimaryNodeState provides context about what changed so that the component can take appropriate action. The PrimaryNodeState enumerator has two possible values: ELECTED_PRIMARY_NODE (the node receiving this state has been elected the Primary Node of the NiFi cluster), or PRIMARY_NODE_REVOKED (the node receiving this state was the Primary Node but has now had its Primary Node role revoked).

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 be 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 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.

Documenting a Component

NiFi attempts to make the user experience as simple and convenient as possible by providing significant amount of documentation to the user from within the NiFi application itself via the User Interface. In order for this to happen, of course, Processor developers must provide that documentation to the framework. NiFi exposes a few different mechanisms for supplying documentation to the framework.

Documenting Properties

Individual properties can be documented by calling the description method of a PropertyDescriptor's builder as such:

```
        public static final PropertyDescriptor MY_PROPERTY = new
PropertyDescriptor.Builder()
 .name("My Property")
 .description("Description of the Property")
 ...
 .build();
```

If the property is to provide a set of allowable values, those values are presented to the user in a drop-down field in the UI. Each of those values can also be given a description:

```
        public static final AllowableValue EXTENSIVE = new
 AllowableValue("Extensive", "Extensive",
        "Everything will be logged - use with caution!");
public static final AllowableValue VERBOSE = new AllowableValue("Verbose",
 "Verbose",
        "Quite a bit of logging will occur");
public static final AllowableValue REGULAR = new AllowableValue("Regular",
 "Regular",
        "Typical logging will occur");

public static final PropertyDescriptor LOG_LEVEL = new
 PropertyDescriptor.Builder()
  .name("Amount to Log")
  .description("How much the Processor should log")
  .allowableValues(REGULAR, VERBOSE, EXTENSIVE)
  .defaultValue(REGULAR.getValue())
  ...
  .build();
```

Documenting Relationships

Processor Relationships are documented in much the same way that properties are - by calling the description method of a Relationship's builder:

```
        public static final Relationship MY_RELATIONSHIP = new
 Relationship.Builder()
  .name("My Relationship")
  .description("This relationship is used only if the Processor fails to
 process the data.")
  .build();
```

Documenting Capability and Keywords

The org.apache.nifi.annotations.documentation package provides Java annotations that can be used to document components. The CapabilityDescription annotation can be added to a Processor, Reporting Task, or Controller Service and is intended to provide a brief description of the functionality provided by the component. The Tags annotation has a value variable that is defined to be an Array of Strings. As such, it is used by providing multiple values as a comma-separated list of Strings with curly braces. These values are then incorporated into the UI by allowing users to filter the components based on a tag (i.e., a keyword). Additionally, the UI provides a tag cloud that allows users to select the tags that they want to filter by. The tags that are largest in the cloud are those tags that exist the most on the components in that instance of NiFi. An example of using these annotations is provided below:

```
        @Tags({"example", "documentation", "developer guide", "processor",
 "tags"})
@CapabilityDescription("Example Processor that provides no real
 functionality but is provided" +
        " for an example in the Developer Guide")
public static final ExampleProcessor extends Processor {
    ...
```

```
}
```

Documenting FlowFile Attribute Interaction

Many times a processor will expect certain FlowFile attributes be set on in-bound FlowFiles in order for the processor to function properly. In other cases a processor may update or create FlowFile attributes on the out-bound FlowFile. Processor developers may document both of these behaviors using the ReadsAttribute and WritesAttribute documentation annotations. These attributes are used to generate documentation that gives users a better understanding of how a processor will interact with the flow.

Note: Because Java 7 does not support repeated annotations on a type, you may need to use ReadsAttributes and WritesAttributes to indicate that a processor reads or writes multiple FlowFile attributes. This annotation can only be applied to Processors. An example is listed below:

```
        @WritesAttributes({ @WritesAttribute(attribute =
 "invokehttp.status.code", description = "The status code that is
 returned"),
       @WritesAttribute(attribute = "invokehttp.status.message",
 description = "The status message that is returned"),
       @WritesAttribute(attribute = "invokehttp.response.body", description
 = "The response body"),
       @WritesAttribute(attribute = "invokehttp.request.url", description =
 "The request URL"),
       @WritesAttribute(attribute = "invokehttp.tx.id", description = "The
 transaction ID that is returned after reading the response"),
       @WritesAttribute(attribute = "invokehttp.remote.dn", description =
 "The DN of the remote server") })
public final class InvokeHTTP extends AbstractProcessor {
```

Documenting Related Components

Often Processors and ControllerServices are related to one another. Sometimes it is a put/get relation as in PutFile and GetFile. Sometimes a Processor uses a ControllerService like InvokeHTTP and StandardSSLContextService. Sometimes one ControllerService uses another like DistributedMapCacheClientService and DistributedMapCacheServer. Developers of these extension points may relate these different components using the SeeAlso tag. This annotation links these components in the documentation. SeeAlso can be applied to Processors, ControllerServices and ReportingTasks. An example of how to do this is listed below:

```
        @SeeAlso(GetFile.class)
public class PutFile extends AbstractProcessor {
```

Advanced Documentation

When the documentation methods above are not sufficient, NiFi provides the ability to expose more advanced documentation to the user via the "Usage" documentation. When a user right-clicks on a Processor, NiFi provides a "Usage" menu item in the context menu. Additionally, the UI exposes a "Help" link in the top-right corner, from which the same Usage information can be found.

The advanced documentation of a Processor is provided as an HTML file named additionalDetails.html. This file should exist within a directory whose name is the fully-qualified name of the Processor, and this directory's parent should be named docs and exist in the root of the Processor's jar. This file will be linked from a generated HTML file that will contain all the Capability, Keyword, PropertyDescription and Relationship information, so it will not be necessary to duplicate that. This is a place to provide a rich explanation of what this Processor is doing, what kind of data it expects and produces, and what FlowFile attributes it expects and produces. Because this documentation is in an HTML format, you may include images and tables to best describe this component. The same methods can be used to provide advanced documentation for Processors, ControllerServices and ReportingTasks.

Provenance Events

The different event types for provenance reporting are:

Provenance Event	Description
ADDINFO	Indicates a provenance event for adding additional information such as new linkage to a new URI or UUID
ATTRIBUTES_MODIFIED	Indicates that a FlowFile's attributes were modified in some way. This event is not needed when another event is reported at the same time, as the other event will already contain all FlowFile attributes
CLONE	Indicates that a FlowFile is an exact duplicate of its parent FlowFile
CONTENT_MODIFIED	Indicates that a FlowFile's content was modified in some way. When using this Event Type, it is advisable to provide details about how the content is modified
CREATE	Indicates that a FlowFile was generated from data that was not received from a remote system or external process
DOWNLOAD	Indicates that the contents of a FlowFile were downloaded by a user or external entity
DROP	Indicates a provenance event for the conclusion of an object's life for some reason other than object expiration
EXPIRE	Indicates a provenance event for the conclusion of an object's life due to the object not being processed in a timely manner
FETCH	Indicates that the contents of a FlowFile were overwritten using the contents of some external resource. This is similar to the RECEIVE event but varies in that RECEIVE events are intended to be used as the event that introduces the FlowFile into the system, whereas FETCH is used to indicate that the contents of an existing FlowFile were overwritten
FORK	Indicates that one or more FlowFiles were derived from a parent FlowFile
JOIN	Indicates that a single FlowFile is derived from joining together multiple parent FlowFiles
RECEIVE	Indicates a provenance event for receiving data from an external process. This Event Type is expected to be the first event for a FlowFile. As such, a Processor that receives data from an external source and uses that data to replace the content of an existing FlowFile should use the FETCH event type, rather than the RECEIVE event type

Provenance Event	Description
REPLAY	Indicates a provenance event for replaying a FlowFile. The UUID of the event indicates the UUID of the original FlowFile that is being replayed. The event contains one Parent UUID that is also the UUID of the FlowFile that is being replayed and one Child UUID that is the UUID of the a newly created FlowFile that will be re-queued for processing
ROUTE	Indicates that a FlowFile was routed to a specified relationship and provides information about why the FlowFile was routed to this relationship
SEND	Indicates a provenance event for sending data to an external process
UNKNOWN	Indicates that the type of provenance event is unknown because the user who is attempting to access the event is not authorized to know the type

Common Processor Patterns

While there are many different Processors available to NiFi users, the vast majority of them fall into one of several common design patterns. Below, we discuss these patterns, when the patterns are appropriate, reasons we follow these patterns, and things to watch out for when applying such patterns. Note that the patterns and recommendations discussed below are general guidelines and not hardened rules.

Data Ingress

A Processor that ingests data into NiFi has a single Relationship named success. This Processor generates new FlowFiles via the ProcessSession create method and does not pull FlowFiles from incoming Connections. The Processor name starts with "Get" or "Listen," depending on whether it polls an external source or exposes some interface to which external sources can connect. The name ends with the protocol used for communications. Processors that follow this pattern include GetFile, GetSFTP, ListenHTTP, and GetHTTP.

This Processor may create or initialize a Connection Pool in a method that uses the @OnScheduled annotation. However, because communications problems may prevent connections from being established or cause connections to be terminated, connections themselves are not created at this point. Rather, the connections are created or leased from the pool in the onTrigger method.

The onTrigger method of this Processor begins by leasing a connection from the Connection Pool, if possible, or otherwise creates a connection to the external service. When no data is available from the external source, the yield method of the ProcessContext is called by the Processor and the method returns so that this Processor avoids continually running and depleting resources without benefit. Otherwise, this Processor then creates a FlowFile via the ProcessSession's create method and assigns an appropriate filename and path to the FlowFile (by adding the filename and path attributes), as well as any other attributes that may be appropriate. An OutputStream to the FlowFile's content is obtained via the ProcessSession's write method, passing a new OutputStreamCallback (which is usually an anonymous inner class). From within this callback, the Processor is able to write to the FlowFile and streams the content from the external resource to the FlowFile's OutputStream. If the desire is to write the entire contents of an InputStream to the FlowFile, the importFrom method of ProcessSession may be more convenient to use than the write method.

When this Processor expects to receive many small files, it may be advisable to create several FlowFiles from a single session before committing the session. Typically, this allows the Framework to treat the content of the newly created FlowFiles much more efficiently.

This Processor generates a Provenance event indicating that it has received data and specifies from where the data came. This Processor should log the creation of the FlowFile so that the FlowFile's origin can be determined by analyzing logs, if necessary.

This Processor acknowledges receipt of the data and/or removes the data from the external source in order to prevent receipt of duplicate files. This is done only after the ProcessSession by which the FlowFile was created has been committed! Failure to adhere to this principle may result in data loss, as restarting NiFi before the session has been committed will result in the temporary file being deleted. Note, however, that it is possible using this approach to receive duplicate data because the application could be restarted after committing the session and before acknowledging or removing the data from the external source. In general, though, potential data duplication is preferred over potential data loss. The connection is finally returned or added to the Connection Pool, depending on whether the connection was leased from the Connection Pool to begin with or was created in the onTrigger method.

If there is a communications problem, the connection is typically terminated and not returned (or added) to the Connection Pool. Connections to remote systems are torn down and the Connection Pool shutdown in a method annotated with the @OnStopped annotation so that resources can be reclaimed.

Data Egress

A Processor that publishes data to an external source has two Relationships: success and failure. The Processor name starts with "Put" followed by the protocol that is used for data transmission. Processors that follow this pattern include PutEmail, PutSFTP, and PostHTTP (note that the name does not begin with "Put" because this would lead to confusion, since PUT and POST have special meanings when dealing with HTTP).

This Processor may create or initialize a Connection Pool in a method that uses the @OnScheduled annotation. However, because communications problems may prevent connections from being established or cause connections to be terminated, connections themselves are not created at this point. Rather, the connections are created or leased from the pool in the onTrigger method.

The onTrigger method first obtains a FlowFile from the ProcessSession via the get method. If no FlowFile is available, the method returns without obtaining a connection to the remote resource.

If at least one FlowFile is available, the Processor obtains a connection from the Connection Pool, if possible, or otherwise creates a new connection. If the Processor is neither able to lease a connection from the Connection Pool nor create a new connection, the FlowFile is routed to failure, the event is logged, and the method returns.

If a connection was obtained, the Processor obtains an InputStream to the FlowFile's content by invoking the read method on the ProcessSession and passing an InputStreamCallback (which is often an anonymous inner class) and from within that callback transmits the contents of the FlowFile to the destination. The event is logged along with the amount of time taken to transfer the file and the data rate at which the file was transferred. A SEND event is reported to the ProvenanceReporter by obtaining the reporter from the ProcessSession via the getProvenanceReporter method and calling the send method on the reporter. The connection is returned or added to the Connection Pool, depending on whether the connection was leased from the pool or newly created by the onTrigger method.

If there is a communications problem, the connection is typically terminated and not returned (or added) to the Connection Pool. If there is an issue sending the data to the remote resource, the desired approach for handling the error depends on a few considerations. If the issue is related to a network condition, the FlowFile is generally routed to failure. The FlowFile is not penalized because there is not necessary a problem with the data. Unlike the case of the Ingress Processor, we typically do not call yield on the ProcessContext. This is because in the case of ingest, the FlowFile does not exist until the Processor is able to perform its function. However, in the case of a Put Processor, the DataFlow Manager may choose to route failure to a different Processor. This can allow for a "backup" system to be used in the case of problems with one system or can be used for load distribution across many systems.

If a problem occurs that is data-related, one of two approaches should be taken. First, if the problem is likely to sort itself out, the FlowFile is penalized and then routed to failure. This is the case, for instance, with PutFTP, when a FlowFile cannot be transferred because of a file naming conflict. The presumption is that the file will eventually be removed from the directory so that the new file can be transferred. As a result, we penalize the FlowFile and route to failure so that we can try again later. In the other case, if there is an actual problem with the data (such as the data does not conform to some required specification), a different approach may be taken. In this case, it may be

advantageous to break apart the failure relationship into a failure and a communications failure relationship. This allows the DataFlow Manager to determine how to handle each of these cases individually. It is important in these situations to document well the differences between the two Relationships by clarifying it in the "description" when creating the Relationship.

Connections to remote systems are torn down and the Connection Pool shutdown in a method annotated with @OnStopped so that resources can be reclaimed.

Route Based on Content (One-to-One)

A Processor that routes data based on its content will take one of two forms: Route an incoming FlowFile to exactly one destination, or route incoming data to 0 or more destinations. Here, we will discuss the first case.

This Processor has two relationships: matched and unmatched. If a particular data format is expected, the Processor will also have a failure relationship that is used when the input is not of the expected format. The Processor exposes a Property that indicates the routing criteria.

If the Property that specifies routing criteria requires processing, such as compiling a Regular Expression, this processing is done in a method annotated with @OnScheduled, if possible. The result is then stored in a member variable that is marked as volatile.

The onTrigger method obtains a single FlowFile. The method reads the contents of the FlowFile via the ProcessSession's read method, evaluating the Match Criteria as the data is streamed. The Processor then determines whether the FlowFile should be routed to matched or unmatched based on whether or not the criteria matched, and routes the FlowFile to the appropriate relationship.

The Processor then emits a Provenance ROUTE event indicating which Relationship to which the Processor routed the FlowFile.

This Processor is annotated with the @SideEffectFree and @SupportsBatching annotations from the org.apache.nifi.annotations.behavior package.

Route Based on Content (One-to-Many)

If a Processor will route a single FlowFile to potentially many relationships, this Processor will be slightly different than the above-described Processor for Routing Data Based on Content. This Processor typically has Relationships that are dynamically defined by the user as well as an unmatched relationship.

In order for the user to be able to define additionally Properties, the getSupportedDynamicPropertyDescriptor method must be overridden. This method returns a PropertyDescriptor with the supplied name and an applicable Validator to ensure that the user-specified Matching Criteria is valid.

In this Processor, the Set of Relationships that is returned by the getRelationships method is a member variable that is marked volatile. This Set is initially constructed with a single Relationship named unmatched. The onPropertyModified method is overridden so that when a Property is added or removed, a new Relationship is created with the same name. If the Processor has Properties that are not user-defined, it is important to check if the specified Property is user-defined. This can be achieved by calling the isDynamic method of the PropertyDescriptor that is passed to this method. If this Property is dynamic, a new Set of Relationships is then created, and the previous set of Relationships is copied into it. This new Set either has the newly created Relationship added to it or removed from it, depending on whether a new Property was added to the Processor or a Property was removed (Property removal is detected by check if the third argument to this function is null). The member variable holding the Set of Relationships is then updated to point to this new Set.

If the Properties that specify routing criteria require processing, such as compiling a Regular Expression, this processing is done in a method annotated with @OnScheduled, if possible. The result is then stored in a member variable that is marked as volatile. This member variable is generally of type Map where the key is of type Relationship and the value's type is defined by the result of processing the property value.

The onTrigger method obtains a FlowFile via the get method of ProcessSession. If no FlowFile is available, it returns immediately. Otherwise, a Set of type Relationship is created. The method reads the contents of the FlowFile via the ProcessSession's read method, evaluating each of the Match Criteria as the data is streamed. For any criteria that matches, the relationship associated with that Match Criteria is added to the Set of Relationships.

After reading the contents of the FlowFile, the method checks if the Set of Relationships is empty. If so, the original FlowFile has an attribute added to it to indicate the Relationship to which it was routed and is routed to the unmatched. This is logged, a Provenance ROUTE event is emitted, and the method returns. If the size of the Set is equal to 1, the original FlowFile has an attribute added to it to indicate the Relationship to which it was routed and is routed to the Relationship specified by the entry in the Set. This is logged, a Provenance ROUTE event is emitted for the FlowFile, and the method returns.

In the event that the Set contains more than 1 Relationship, the Processor creates a clone of the FlowFile for each Relationship, except for the first. This is done via the clone method of the ProcessSession. There is no need to report a CLONE Provenance Event, as the framework will handle this for you. The original FlowFile and each clone are routed to their appropriate Relationship with attribute indicating the name of the Relationship. A Provenance ROUTE event is emitted for each FlowFile. This is logged, and the method returns.

This Processor is annotated with the @SideEffectFree and @SupportsBatching annotations from the org.apache.nifi.annotations.behavior package.

Route Streams Based on Content (One-to-Many)

The previous description of Route Based on Content (One-to-Many) provides an abstraction for creating a very powerful Processor. However, it assumes that each FlowFile will be routed in its entirety to zero or more Relationships. What if the incoming data format is a "stream" of many different pieces of information - and we want to send different pieces of this stream to different Relationships? For example, imagine that we want to have a RouteCSV Processor such that it is configured with multiple Regular Expressions. If a line in the CSV file matches a Regular Expression, that line should be included in the outbound FlowFile to the associated relationship. If a Regular Expression is associated with the Relationship "has-apples" and that Regular Expression matches 1,000 of the lines in the FlowFile, there should be one outbound FlowFile for the "has-apples" relationship that has 1,000 lines in it. If a different Regular Expression is associated with the Relationship "has-oranges" and that Regular Expression matches 50 lines in the FlowFile, there should be one outbound FlowFile for the "has-oranges" relationship that has 50 lines in it. I.e., one FlowFile comes in and two FlowFiles come out. The two FlowFiles may contain some of the same lines of text from the original FlowFile, or they may be entirely different. This is the type of Processor that we will discuss in this section.

This Processor's name starts with "Route" and ends with the name of the data type that it routes. In our example here, we are routing CSV data, so the Processor is named RouteCSV. This Processor supports dynamic properties. Each user-defined property has a name that maps to the name of a Relationship. The value of the Property is in the format necessary for the "Match Criteria." In our example, the value of the property must be a valid Regular Expression.

This Processor maintains an internal ConcurrentMap where the key is a Relationship and the value is of a type dependent on the format of the Match Criteria. In our example, we would maintain a ConcurrentMap<Relationship, Pattern>. This Processor overrides the onPropertyModified method. If the new value supplied to this method (the third argument) is null, the Relationship whose name is defined by the property name (the first argument) is removed from the ConcurrentMap. Otherwise, the new value is processed (in our example, by calling Pattern.compile(newValue)) and this value is added to the ConcurrentMap with the key again being the Relationship whose name is specified by the property name.

This Processor will override the customValidate method. In this method, it will retrieve all Properties from the ValidationContext and count the number of PropertyDescriptors that are dynamic (by calling isDynamic() on the PropertyDescriptor). If the number of dynamic PropertyDescriptors is 0, this indicates that the user has not added any Relationships, so the Processor returns a ValidationResult indicating that the Processor is not valid because it has no Relationships added.

The Processor returns all of the Relationships specified by the user when its getRelationships method is called and will also return an unmatched Relationship. Because this Processor will have to read and write to the Content

Repository (which can be relatively expensive), if this Processor is expected to be used for very high data volumes, it may be advantageous to add a Property that allows the user to specify whether or not they care about the data that does not match any of the Match Criteria.

When the onTrigger method is called, the Processor obtains a FlowFile via ProcessSession.get. If no data is available, the Processor returns. Otherwise, the Processor creates a Map<Relationship, FlowFile>. We will refer to this Map as flowFileMap. The Processor reads the incoming FlowFile by calling ProcessSession.read and provides an InputStreamCallback. From within the Callback, the Processor reads the first piece of data from the FlowFile. The Processor then evaluates each of the Match Criteria against this piece of data. If a particular criteria (in our example, a Regular Expression) matches, the Processor obtains the FlowFile from flowFileMap that belongs to the appropriate Relationship. If no FlowFile yet exists in the Map for this Relationship, the Processor creates a new FlowFile by calling session.create(incomingFlowFile) and then adds the new FlowFile to flowFileMap. The Processor then writes this piece of data to the FlowFile by calling session.append with an OutputStreamCallback. From within this OutputStreamCallback, we have access to the new FlowFile's OutputStream, so we are able to write the data to the new FlowFile. We then return from the OutputStreamCallback. After iterating over each of the Match Criteria, if none of them match, we perform the same routines as above for the unmatched relationship (unless the user configures us to not write out unmatched data). Now that we have called session.append, we have a new version of the FlowFile. As a result, we need to update our flowFileMap to associate the Relationship with the new FlowFile.

If at any point, an Exception is thrown, we will need to route the incoming FlowFile to failure. We will also need to remove each of the newly created FlowFiles, as we won't be transferring them anywhere. We can accomplish this by calling session.remove(flowFileMap.values()). At this point, we will log the error and return.

Otherwise, if all is successful, we can now iterate through the flowFileMap and transfer each FlowFile to the corresponding Relationship. The original FlowFile is then either removed or routed to an original relationship. For each of the newly created FlowFiles, we also emit a Provenance ROUTE event indicating which Relationship the FlowFile went to. It is also helpful to include in the details of the ROUTE event how many pieces of information were included in this FlowFile. This allows DataFlow Managers to easily see when looking at the Provenance Lineage view how many pieces of information went to each of the relationships for a given input FlowFile.

Additionally, some Processors may need to "group" the data that is sent to each Relationship so that each FlowFile that is sent to a relationship has the same value. In our example, we may wan to allow the Regular Expression to have a Capturing Group and if two different lines in the CSV match the Regular Expression but have different values for the Capturing Group, we want them to be added to two different FlowFiles. The matching value could then be added to each FlowFile as an Attribute. This can be accomplished by modifying the flowFileMap such that it is defined as Map<Relationship, Map<T, FlowFile>> where T is the type of the Grouping Function (in our example, the Group would be a String because it is the result of evaluating a Regular Expression's Capturing Group).

Route Based on Attributes

This Processor is almost identical to the Route Data Based on Content Processors described above. It takes two different forms: One-to-One and One-to-Many, as do the Content-Based Routing Processors. This Processor, however, does not make any call to ProcessSession's read method, as it does not read FlowFile content. This Processor is typically very fast, so the @SupportsBatching annotation can be very important in this case.

Split Content (One-to-Many)

This Processor generally requires no user configuration, with the exception of the size of each Split to create. The onTrigger method obtains a FlowFile from its input queues. A List of type FlowFile is created. The original FlowFile is read via the ProcessSession's read method, and an InputStreamCallback is used. Within the InputStreamCallback, the content is read until a point is reached at which the FlowFile should be split. If no split is needed, the Callback returns, and the original FlowFile is routed to success. In this case, a Provenance ROUTE event is emitted. Typically, ROUTE events are not emitted when routing a FlowFile to success because this generates a very verbose lineage that becomes difficult to navigate. However, in this case,the event is useful because we would otherwise expect a

FORK event and the absence of any event is likely to cause confusion. The fact that the FlowFile was not split but was instead transferred to success is logged, and the method returns.

If a point is reached at which a FlowFile needs to be split, a new FlowFile is created via the ProcessSession's create(FlowFile) method or the clone(FlowFile, long, long) method. The next section of code depends on whether the create method is used or the clone method is used. Both methods are described below. Which solution is appropriate must be determined on a case-by-case basis.

The Create Method is most appropriate when the data will not be directly copied from the original FlowFile to the new FlowFile. For example, if only some of the data will be copied, or if the data will be modified in some way before being copied to the new FlowFile, this method is necessary. However, if the content of the new FlowFile will be an exact copy of a portion of the original FlowFile, the Clone Method is much preferred.

Create Method If using the create method, the method is called with the original FlowFile as the argument so that the newly created FlowFile will inherit the attributes of the original FlowFile and a Provenance FORK event will be created by the framework.

The code then enters a try/finally block. Within the finally block, the newly created FlowFile is added to the List of FlowFiles that have been created. This is done within a finally block so that if an Exception is thrown, the newly created FlowFile will be appropriately cleaned up. Within the try block, the callback initiates a new callback by calling the ProcessSession's write method with an OutputStreamCallback. The appropriate data is then copied from the InputStream of the original FlowFile to the OutputStream for the new FlowFile.

Clone Method If the content of the newly created created FlowFile is to be only a contiguous subset of the bytes of the original FlowFile, it is preferred to use the clone(FlowFile, long, long) method instead of the create(FlowFile) method of the ProcessSession. In this case, the offset of the original FlwoFile at which the new FlowFile's content should begin is passed as the second argument to the clone method. The length of the new FlowFile is passed as the third argument to the clone method. For example, if the original FlowFile was 10,000 bytes and we called clone(flowFile, 500, 100), the FlowFile that would be returned to us would be identical to flowFile with respect to its attributes. However, the content of the newly created FlowFile would be 100 bytes in length and would start at offset 500 of the original FlowFile. That is, the contents of the newly created FlowFile would be the same as if you had copied bytes 500 through 599 of the original FlowFile.

After the clone has been created, it is added to the List of FlowFiles.

This method is much more highly preferred than the Create method, when applicable, because no disk I/O is required. The framework is able to simply create a new FlowFile that references a subset of the original FlowFile's content, rather than actually copying the data. However, this is not always possible. For example, if header information must be copied from the beginning of the original FlowFile and added to the beginning of each Split, then this method is not possible.

Both Methods Regardless of whether the Clone Method or the Create Method is used, the following is applicable:

If at any point in the InputStreamCallback, a condition is reached in which processing cannot continue (for example, the input is malformed), a ProcessException should be thrown. The call to the ProcessSession's read method is wrapped in a try/catch block where ProcessException is caught. If an Exception is caught, a log message is generated explaining the error. The List of newly created FlowFiles is removed via the ProcessSession's remove method. The original FlowFile is routed to failure.

If no problems arise, the original FlowFile is routed to original and all newly created FlowFiles are updated to include the following attributes:

Attribute Name	Description
split.parent.uuid	The UUID of the original FlowFile
split.index	A one-up number indicating which FlowFile in the list this is (the first FlowFile created will have a value 0, the second will have a value 1, etc.)
split.count	The total number of split FlowFiles that were created

The newly created FlowFiles are routed to success; this event is logged; and the method returns.

Update Attributes Based on Content

This Processor is very similar to the Route Based on Content Processors discussed above. Rather than routing a FlowFile to matched or unmatched, the FlowFile is generally routed to success or failure and attributes are added to the FlowFile as appropriate. The attributes to be added are configured in a manner similar to that of the Route Based on Content (One-to-Many), with the user defining their own properties. The name of the property indicates the name of an attribute to add. The value of the property indicates some Matching Criteria to be applied to the data. If the Matching Criteria matches the data, an attribute is added with the name the same as that of the Property. The value of the attribute is the criteria from the content that matched.

For example, a Processor that evaluates XPath Expressions may allow user-defined XPaths to be entered. If the XPath matches the content of a FlowFile, that FlowFile will have an attribute added with the name being equal to that of the Property name and a value equal to the textual content of the XML Element or Attribute that matched the XPath. The failure relationship would then be used if the incoming FlowFile was not valid XML in this example. The success relationship would be used regardless of whether or not any matches were found. This can then be used to route the FlowFile when appropriate.

This Processor emits a Provenance Event of type ATTRIBUTES_MODIFIED.

Enrich/Modify Content

The Enrich/Modify Content pattern is very common and very generic. This pattern is responsible for any general content modification. For the majority of cases, this Processor is marked with the @SideEffectFree and @SupportsBatching annotations. The Processor has any number of required and optional Properties, depending on the Processor's function. The Processor generally has a success and failure relationship. The failure relationship is generally used when the input file is not in the expected format.

This Processor obtains a FlowFile and updates it using the ProcessSession's write(StreamCallback) method so that it is able to both read from the FlowFile's content and write to the next version of the FlowFile's content. If errors are encountered during the callback, the callback will throw a ProcessException. The call to the ProcessSession's write method is wrapped in a try/catch block that catches ProcessException and routes the FlowFile to failure.

If the callback succeeds, a CONTENT_MODIFIED Provenance Event is emitted.

Error Handling

When writing a Processor, there are several different unexpected cases that can occur. It is important that Processor developers understand the mechanics of how the NiFi framework behaves if Processors do not handle errors themselves, and it's important to understand what error handling is expected of Processors. Here, we will discuss how Processors should handle unexpected errors during the course of their work.

Exceptions within the Processor

During the execution of the onTrigger method of a Processor, many things can potentially go awry. Common failure conditions include:

- Incoming data is not in the expected format.
- Network connections to external services fail.
- Reading or writing data to a disk fails.
- There is a bug in the Processor or a dependent library.

Any of these conditions can result in an Exception being thrown from the Processor. From the framework perspective, there are two types of Exceptions that can escape a Processor: ProcessException and all others.

If a ProcessException is thrown from the Processor, the framework will assume that this is a failure that is a known outcome. Moreover, it is a condition where attempting to process the data again later may be successful. As a result, the framework will roll back the session that was being processed and penalize the FlowFiles that were being processed.

If any other Exception escapes the Processor, though, the framework will assume that it is a failure that was not taken into account by the developer. In this case, the framework will also roll back the session and penalize the FlowFiles. However, in this case, we can get into some very problematic cases. For example, the Processor may be in a bad state and may continually run, depleting system resources, without providing any useful work. This is fairly common, for instance, when a NullPointerException is thrown continually. In order to avoid this case, if an Exception other than ProcessException is able to escape the Processor's onTrigger method, the framework will also "Administratively Yield" the Processor. This means that the Processor will not be triggered to run again for some amount of time. The amount of time is configured in the nifi.properties file but is 10 seconds by default.

Exceptions within a callback: IOException, RuntimeException

More often than not, when an Exception occurs in a Processor, it occurs from within a callback (I.e., InputStreamCallback, OutputStreamCallback, or StreamCallback). That is, during the processing of a FlowFile's content. Callbacks are allowed to throw either RuntimeException or IOException. In the case of RuntimeException, this Exception will propagate back to the onTrigger method. In the case of an IOException, the Exception will be wrapped within a ProcessException and this ProcessException will then be thrown from the Framework.

For this reason, it is recommended that Processors that use callbacks do so within a try/catch block and catch ProcessException as well as any other RuntimeException that they expect their callback to throw. It is not recommended that Processors catch the general Exception or Throwable cases, however. This is discouraged for two reasons.

First, if an unexpected RuntimeException is thrown, it is likely a bug and allowing the framework to rollback the session will ensure no data loss and ensures that DataFlow Managers are able to deal with the data as they see fit by keeping the data queued up in place.

Second, when an IOException is thrown from a callback, there really are two types of IOExceptions: those thrown from Processor code (for example, the data is not in the expected format or a network connection fails), and those that are thrown from the Content Repository (where the FlowFile content is stored). If the latter is the case, the framework will catch this IOException and wrap it into a FlowFileAccessException, which extends RuntimeException. This is done explicitly so that the Exception will escape the onTrigger method and the framework can handle this condition appropriately. Catching the general Exception prevents this from happening.

Penalization vs. Yielding

When an issue occurs during processing, the framework exposes two methods to allow Processor developers to avoid performing unnecessary work: "penalization" and "yielding." These two concepts can become confusing for developers new to the NiFi API. A developer is able to penalize a FlowFile by calling the penalize(FlowFile) method of ProcessSession. This causes the FlowFile itself to be inaccessible to downstream Processors for a period of time. The amount of time that the FlowFile is inaccessible is determined by the DataFlow Manager by setting the "Penalty Duration" setting in the Processor Configuration dialog. The default value is 30 seconds. Typically, this is done when a Processor determines that the data cannot be processed due to environmental reasons that are expected to sort themselves out. A great example of this is the PutSFTP processor, which will penalize a FlowFile if a file already exists on the SFTP server that has the same filename. In this case, the Processor penalizes the FlowFile and routes it to failure. A DataFlow Manager can then route failure back to the same PutSFTP Processor. This way, if a file exists with the same filename, the Processor will not attempt to send the file again for 30 seconds (or whatever period the DFM has configured the Processor to use). In the meantime, it is able to continue to process other FlowFiles.

On the other hand, yielding allows a Processor developer to indicate to the framework that it will not be able to perform any useful function for some period of time. This commonly happens with a Processor that is communicating with a remote resource. If the Processor cannot connect to the remote resource, or if the remote resource is expected to provide data but reports that it has none, the Processor should call yield on the ProcessContext object and then return. By doing this, the Processor is telling the framework that it should not waste resources triggering this Processor to run, because there's nothing that it can do - it's better to use those resources to allow other Processors to run.

Session Rollback

Thus far, when we have discussed the ProcessSession, we have typically referred to it simply as a mechanism for accessing FlowFiles. However, it provides another very important capability, which is transactionality. All methods that are called on a ProcessSession happen as a transaction. When we decided to end the transaction, we can do so either by calling commit() or by calling rollback(). Typically, this is handled by the AbstractProcessor class: if the onTrigger method throws an Exception, the AbstractProcessor will catch the Exception, call session.rollback(), and then re-throw the Exception. Otherwise, the AbstractProcessor will call commit() on the ProcessSession.

There are times, however, that developers will want to roll back a session explicitly. This can be accomplished at any time by calling the rollback() or rollback(boolean) method. If using the latter, the boolean indicates whether or not those FlowFiles that have been pulled from queues (via the ProcessSession get methods) should be penalized before being added back to their queues.

When rollback is called, any modification that has occurred to the FlowFiles in that session are discarded, to included both content modification and attribute modification. Additionally, all Provenance Events are rolled back (with the exception of any SEND event that was emitted by passing a value of true for the force argument). The FlowFiles that were pulled from the input queues are then transferred back to the input queues (and optionally penalized) so that they can be processed again.

On the other hand, when the commit method is called, the FlowFile's new state is persisted in the FlowFile Repository, and any Provenance Events that occurred are persisted in the Provenance Repository. The previous content is destroyed (unless another FlowFile references the same piece of content), and the FlowFiles are transferred to the outbound queues so that the next Processors can operate on the data.

It is also important to note how this behavior is affected by using the org.apache.nifi.annotations.behavior.SupportsBatching annotation. If a Processor utilizes this annotation, calls to ProcessSession.commit may not take affect immediately. Rather, these commits may be batched together in order to provide higher throughput. However, if at any point, the Processor rolls back the ProcessSession, all changes since the last call to commit will be discarded and all "batched" commits will take affect. These "batched" commits are not rolled back.

General Design Considerations

When designing a Processor, there are a few important design considering to keep in mind. This section of the Developer Guide brings to the forefront some of the ideas that a developer should be thinking about when creating a Processor.

Consider the User

One of the most important concepts to keep in mind when developing a Processor (or any other component) is the user experience that you are creating. It's important to remember that as the developer of such a component, you may have important knowledge about the context that others do not have. Documentation should always be supplied so that those less familiar with the process are able to use it with ease.

When thinking about the user experience, it is also important to note that consistency is very important. It is best to stick with the standard naming conventions. This is true for Processor names, Property names and value, Relationship names, and any other aspect that the user will experience.

Simplicity is crucial! Avoid adding properties that you don't expect users to understand or change. As developers, we are told that hard-coding values is bad. But this sometimes results in developers exposing properties that, when asked for clarification, tell users to just leave the default value. This leads to confusion and complexity.

Cohesion and Reusability

For the sake of making a single, cohesive unit, developers are sometimes tempted to combine several functions into a single Processor. This is very true for the case when a Processor expects input data to be in format X so that the Processor can convert the data into format Y and send the newly-formatted data to some external service.

Taking this approach of formatting the data for a particular endpoint and then sending the data to that endpoint within the same Processor has several drawbacks:

- The Processor becomes very complex, as it has to perform the data translation task as well as the task of sending the data to the remote service.
- If the Processor is unable to communicate with the remote service, it will route the data to a failure Relationship. In this case, the Processor will be responsible to perform the data translation again. And if it fails again, the translation is done yet again.
- If we have five different Processors that translate the incoming data into this new format before sending the data, we have a great deal of duplicated code. If the schema changes, for instance, many Processors must be updated.
- This intermediate data is thrown away when the Processor finishes sending to the remote service. The intermediate data format may well be useful to other Processors.

In order to avoid these issues, and make Processors more reusable, a Processor should always stick to the principal of "do one thing and do it well." Such a Processor should be broken into two separate Processors: one to convert the data from Format X to Format Y, and another Processor to send data to the remote resource.

Naming Conventions

In order to deliver a consistent look and feel to users, it is advisable that Processors keep with standard naming conventions. The following is a list of standard conventions that are used:

- Processors that pull data from a remote system are named Get<Service> or Get<Protocol>, depending on if they poll data from arbitrary sources over a known Protocol (such as GetHTTP or GetFTP) or if they pull data from a known service (such as GetKafka)
- Processors that push data to a remote system are named Put<Service> or Put<Protocol>.
- Relationship names are lower-cased and use spaces to delineated words.
- Property names capitalize significant words, as would be done with the title of a book.

Processor Behavior Annotations

When creating a Processor, the developer is able to provide hints to the framework about how to utilize the Processor most effectively. This is done by applying annotations to the Processor's class. The annotations that can be applied to a Processor exist in three sub-packages of org.apache.nifi.annotations. Those in the documentation sub-package are used to provide documentation to the user. Those in the lifecycle sub-package instruct the framework which methods should be called on the Processor in order to respond to the appropriate life-cycle events. Those in the behavior package help the framework understand how to interact with the Processor in terms of scheduling and general behavior.

The following annotations from the org.apache.nifi.annotations.behavior package can be used to modify how the framework will handle your Processor:

- EventDriven: Instructs the framework that the Processor can be scheduled using the Event-Driven scheduling strategy. This strategy is still experimental at this point, but can result in reduced resource utilization on dataflows that do not handle extremely high data rates.
- SideEffectFree: Indicates that the Processor does not have any side effects external to NiFi. As a result, the framework is free to invoke the Processor many times with the same input without causing any unexpected results to occur. This implies idempotent behavior. This can be used by the framework to improve efficiency by performing actions such as transferring a ProcessSession from one Processor to another, such that if a problem occurs many Processors' actions can be rolled back and performed again.
- SupportsBatching: This annotation indicates that it is okay for the framework to batch together multiple ProcessSession commits into a single commit. If this annotation is present, the user will be able to choose whether they prefer high throughput or lower latency in the Processor's Scheduling tab. This annotation should be applied to most Processors, but it comes with a caveat: if the Processor calls ProcessSession.commit, there is no guarantee that the data has been safely stored in NiFi's Content, FlowFile, and Provenance Repositories. As a result, it is not appropriate for those Processors that receive data from an external source, commit the session, and then delete the remote data or confirm a transaction with a remote resource.
- TriggerSerially: When this annotation is present, the framework will not allow the user to schedule more than one concurrent thread to execute the onTrigger method at a time. Instead, the number of thread ("Concurrent Tasks") will always be set to 1. This does not, however, mean that the Processor does not have to be thread-safe, as the thread that is executing onTrigger may change between invocations.
- PrimaryNodeOnly: Apache NiFi, when clustered, offers two modes of execution for Processors: "Primary Node" and "All Nodes". Although running in all the nodes offers better parallelism, some Processors are known to cause unintended behaviors when run in multiple nodes. For instance, some Processors list or read files from remote filesystems. If such Processors are scheduled to run on "All Nodes", it will cause unnecessary duplication and even errors. Such Processors should use this annotation. Applying this annotation will restrict the Processor to run only on the "Primary Node".
- TriggerWhenAnyDestinationAvailable: By default, NiFi will not schedule a Processor to run if any of its outbound queues is full. This allows back-pressure to be applied all the way a chain of Processors. However, some Processors may need to run even if one of the outbound queues is full. This annotations indicates that the Processor should run if any Relationship is "available." A Relationship is said to be "available" if none of the connections that use that Relationship is full. For example, the DistributeLoad Processor makes use of this annotation. If the "round robin" scheduling strategy is used, the Processor will not run if any outbound queue is full. However, if the "next available" scheduling strategy is used, the Processor will run if any Relationship at all is available and will route FlowFiles only to those relationships that are available.
- TriggerWhenEmpty: The default behavior is to trigger a Processor to run only if its input queue has at least one FlowFile or if the Processor has no input queues (which is typical of a "source" Processor). Applying this annotation will cause the framework to ignore the size of the input queues and trigger the Processor regardless of whether or not there is any data on an input queue. This is useful, for example, if the Processor needs to be triggered to run periodically to time out a network connection.
- InputRequirement: By default, all Processors will allow users to create incoming connections for the Processor, but if the user does not create an incoming connection, the Processor is still valid and can be scheduled to run. For Processors that are expected to be used as a "Source Processor," though, this can be confusing to the user, and the user may attempt to send FlowFiles to that Processor, only for the FlowFiles to queue up without being processed. Conversely, if the Processor expects incoming FlowFiles but does not have an input queue, the Processor will be scheduled to run but will perform no work, as it will receive no FlowFile, and this leads to confusion as well. As a result, we can use the @InputRequirement annotation and provide it a value of INPUT_REQUIRED, INPUT_ALLOWED, or INPUT_FORBIDDEN. This provides information to the framework about when the Processor should be made invalid, or whether or not the user should even be able to draw a Connection to the Processor. For instance, if a Processor is annotated with InputRequirement(Requirement.INPUT_FORBIDDEN), then the user will not even be able to create a Connection with that Processor as the destination.

Data Buffering

An important point to keep in mind is that NiFi provides a generic data processing capability. Data can be in any format. Processors are generally scheduled with several threads. A common mistake that developers new to NiFi make is to buffer all the contents of a FlowFile in memory. While there are cases when this is required, it should be avoided if at all possible, unless it is well-known what format the data is in. For example, a Processor responsible for executing XPath against an XML document will need to load the entire contents of the data into memory. This is generally acceptable, as XML is not expected to be extremely large. However, a Processor that searches for a specific byte sequence may be used to search files that are hundreds of gigabytes or more. Attempting to load this into memory can cause a lot of problems - especially if multiple threads are processing different FlowFiles simultaneously.

Instead of buffering this data into memory, it is advisable to instead evaluate the data as it is streamed from the Content Repository (i.e., scan the content from the InputStream that is provided to your callback by ProcessSession.read). Of course, in this case, we don't want to read from the Content Repository for each byte, so we would use a BufferedInputStream or somehow buffer some small amount of data, as appropriate.

Controller Services

The ControllerService interface allows developers to share functionality and state across the JVM in a clean and consistent manner. The interface resembles that of the Processor interface but does not have an onTrigger method because Controller Services are not scheduled to run periodically, and Controller Services do not have Relationships because they are not integrated into the flow directly. Rather, they are used by Processors, Reporting Tasks, and other Controller Services.

Developing a ControllerService

Just like with the Processor interface, the ControllerService interface exposes methods for configuration, validation, and initialization. These methods are all identical to those of the Processor interface except that the initialize method is passed a ControllerServiceInitializationContext, rather than a ProcessorInitializationContext.

Controller Services come with an additional constraint that Processors do not have. A Controller Service must be comprised of an interface that extends ControllerService. Implementations can then be interacted with only through their interface. A Processor, for instance, will never be given a concrete implementation of a ControllerService and therefore must reference the service only via interfaces that extends ControllerService.

This constraint is in place mainly because a Processor can exist in one NiFi Archive (NAR) while the implementation of the Controller Service that the Processor lives in can exist in a different NAR. This is accomplished by the framework by dynamically implementing the exposed interfaces in such a way that the framework can switch to the appropriate ClassLoader and invoke the desired method on the concrete implementation. However, in order to make this work, the Processor and the Controller Service implementation must share the same definition of the Controller Service interface. Therefore, both of these NARs must depend on the NAR that houses the Controller Service's interface.

Interacting with a ControllerService

ControllerServices may be obtained by a Processor, another ControllerService, or a ReportingTask by means of the ControllerServiceLookup or by using the identifiesControllerService method of the PropertyDescriptor's Builder class. The ControllerServiceLookup can be obtained by a Processor from the ProcessorInitializationContext that is passed to the initialize method. Likewise, it is obtained by a ControllerService from the ControllerServiceInitializationContext and by a ReportingTask via the ReportingConfiguration object passed to the initialize method.

For most use cases, though, using the identifiesControllerService method of a PropertyDescriptor Builder is preferred and is the least complicated method. In order to use this method, we create a PropertyDescriptor that references a Controller Service as such:

```
        public static final PropertyDescriptor SSL_CONTEXT_SERVICE = new
PropertyDescriptor.Builder()
 .name("SSL Context Service")
 .description("Specified the SSL Context Service that can be used to create
secure connections")
 .required(true)
 .identifiesControllerService(SSLContextService.class)
 .build();
```

Using this method, the user will be prompted to supply the SSL Context Service that should be used. This is done by providing the user with a drop-down menu from which they are able to choose any of the SSLContextService configurations that have been configured, regardless of the implementation.

In order to make use of this service, the Processor can use code such as:

```
        final SSLContextService sslContextService =
context.getProperty(SSL_CONTEXT_SERVICE)
       .asControllerService(SSLContextService.class);
```

Note here that SSLContextService is an interface that extends ControllerService. The only implementation at this time is the StandardSSLContextService. However, the Processor developer need not worry about this detail.

Reporting Tasks

So far, we have mentioned little about how to convey to the outside world how NiFi and its components are performing. Is the system able to keep up with the incoming data rate? How much more can the system handle? How much data is processed at the peak time of day versus the least busy time of day?

In order to answer these questions, and many more, NiFi provides a capability for reporting status, statistics, metrics, and monitoring information to external services by means of the ReportingTask interface. ReportingTasks are given access to a host of information to determine how the system is performing.

Developing a Reporting Task

Just like with the Processor and ControllerService interfaces, the ReportingTask interface exposes methods for configuration, validation, and initialization. These methods are all identical to those of the Processor and ControllerService interfaces except that the initialize method is passed a ReportingConfiguration object, as opposed to the initialization objects received by the other Components. The ReportingTask also has an onTrigger method that is invoked by the framework to trigger the task to perform its job.

Within the onTrigger method, the ReportingTask is given access to a ReportingContext, from which configuration and information about the NiFi instance can be obtained. The BulletinRepository allows Bulletins to be queried and allows the ReportingTask to submit its own Bulletins, so that information will be rendered to users. The ControllerServiceLookup that is accessible via the Context provides access to ControllerServices that have been configured. However, this method of obtaining Controller Services is not the preferred method. Rather, the preferred method for obtaining a Controller Service is to reference the Controller Service in a PropertyDescriptor.

The EventAccess object that is exposed via the ReportingContext provides access to the ProcessGroupStatus, which exposes statistics about the amount of data processed in the past five minutes by Process Groups, Processors, Connections, and other Components. Additionally, the EventAccess object provides access to the ProvenanceEventRecords that have been stored in the ProvenanceEventRepository. These Provenance Events are emitted by Processors when data is received from external sources, emitted to external services, removed from the system, modified, or routed according to some decision that was made.

Each ProvenanceEvent has the ID of the FlowFile, the type of Event, the creation time of the Event, and all FlowFile attributes associated with the FlowFile at the time that the FlowFile was accessed by the component as well as the FlowFile attributes that were associated with the FlowFile as a result of the processing that the event describes. This provides a great deal of information to ReportingTasks, allowing reports to be generated in many different ways to expose metrics and monitoring capabilities needed for any number of operational concerns.

UI Extensions

There are two UI extension points that are available in NiFi:

- Custom Processor UIs
- Content Viewers

Custom UIs can be created to provide configuration options beyond the standard property/value tables available in most processor settings. Examples of processors with Custom UIs are https://github.com/apache/nifi/tree/master/nifi-nar-bundles/nifi-update-attribute-bundle and https://github.com/apache/nifi/tree/master/nifi-nar-bundles/nifi-standard-bundle.

Content Viewers can be created to extend the types of data that can be viewed within NiFi. NiFi comes with NARs in the lib directory which contain content viewers for data types such as csv, xml, avro, json (standard-nar) and image types such as png, jpeg and gif (media-nar).

Custom Processor UIs

To add a Custom UI to a processor:

1. Create your UI.
2. Build and bundle your WAR in a processor NAR.
3. The WAR needs to contain a nifi-processor-configuration file in the META-INF directory, which associates the Custom UI with that processor.
4. Place the NAR in the lib directory and it will be discovered when NiFi starts up.
5. In the Configure Processor window for the processor, the Properties tab should now have an Advanced button, which will access the Custom UI.

As an example, here is the NAR layout for UpdateAttribute:

Update Attribute NAR Layout

```
nifi-update-attribute-bundle
#
### nifi-update-attribute-model
#
### nifi-update-attribute-nar
#
### nifi-update-attribute-processor
#
### nifi-update-attribute-ui
#   ### pom.xml
#   ### src
#       ### main
```

```
#           ### java
#           ### resources
#           ### webapp
#               ### css
#               ### images
#               ### js
#               ### META-INF
#               #   ### nifi-processor-configuration
#               ### WEB-INF
#
### pom.xml
```

with the contents of the nifi-processor-configuration as follows:

org.apache.nifi.processors.attributes.UpdateAttribute:${project.groupId}:nifi-update-attribute-nar:${project.version}

Note: Custom UIs can also be implemented for Controller Services and Reporting Tasks.

Content Viewers

To add a Content Viewer:

1. Build and bundle your WAR in a processor NAR.
2. The WAR needs to contain a nifi-content-viewer file in the META-INF directory, which lists the supported content types.
3. Place the NAR in the lib directory and it will be discovered when NiFi starts up.
4. When a matching content type is encountered, the content viewer will generate the appropriate view.

A good example to follow is the NAR layout for the Standard Content Viewer:

Standard Content Viewer NAR Layout

```
nifi-standard-bundle
#
### nifi-jolt-transform-json-ui
#
### nifi-standard-content-viewer
#   ### pom.xml
#   ### src
#       ### main
#           ### java
#           ### resources
#           ### webapp
#               ### css
#               ### META-INF
#               #   ### nifi-content-viewer
#               ### WEB-INF
#
### nifi-standard-nar
#
### nifi-standard-prioritizers
#
### nifi-standard-processors
#
### nifi-standard-reporting-tasks
#
### nifi-standard-utils
#
### pom.xml
```

with the contents of nifi-content-viewer as follows:

```
application/xml
application/json
text/plain
text/csv
avro/binary
application/avro-binary
application/avro+binary
```

Command Line Tools

The Client/Server mode of operation came about from the desire to automatically generate required TLS configuration artifacts without needing to perform that generation in a centralized place. This simplifies configuration in a clustered environment. Since we don't necessarily have a central place to run the generation logic or a trusted Certificate Authority, a shared secret is used to authenticate the clients and server to each other.

The tls-toolkit prevents man in the middle attacks using HMAC verification of the public keys of the CA server and the CSR the client sends. A shared secret (the token) is used as the HMAC key.

The basic process goes as follows:

1. The client generates a KeyPair.
2. The client generates a request json payload containing a CSR and an HMAC with the token as the key and the CSR's public key fingerprint as the data.
3. The client connects to the CA Hostname at the https port specified and validates that the CN of the CA's certificate matches the hostname (NOTE: because we don't trust the CA at this point, this adds NO security, it is just a way to error out early if possible).
4. The server validates the HMAC from the client payload using the token as the key and the CSR's public key fingerprint as the data. This proves that the client knows the shared secret and that it wanted a CSR with that public key to be signed. (NOTE: a man in the middle could forward this on but wouldn't be able to change the CSR without invalidating the HMAC, defeating the purpose).
5. The server signs the CSR and sends back a response json payload containing the certificate and an HMAC with the token as the key and a fingerprint of its public key as the data.
6. The client validates the response HMAC using the token as the key and a fingerprint of the certificate public key supplied by the TLS session. This validates that a CA that knows the shared secret is the one we are talking to over TLS.
7. The client verifies that the CA certificate from the TLS session signed the certificate in the payload.
8. The client adds the generated KeyPair to its keystore with the certificate chain and adds the CA certificate from the TLS connection to its truststore.
9. The client writes out the configuration json containing keystore, truststore passwords and other details about the exchange.

Testing

Testing the components that will be used within a larger framework can often be very cumbersome and tricky. With NiFi, we strive to make testing components as easy as possible. In order to do this, we have created a nifi-mock module that can be used in conjunction with JUnit to provide extensive testing of components.

The Mock Framework is mostly aimed at testing Processors, as these are by far the most commonly developed extension point. However, the framework does provide the ability to test Controller Services as well.

Components have typically been tested by creating functional tests to verify component behavior. This is done because often a Processor will consist of a handful of helper methods but the logic will largely be encompassed within the onTrigger method. The TestRunner interface allows us to test Processors and Controller Services by converting more "primitive" objects such as files and byte arrays into FlowFiles and handles creating the ProcessSessions and ProcessContexts needed for a Processor to do its job, as well as invoking the necessary lifecycle methods in order to ensure that the Processor behaves the same way in the unit tests as it does in production.

Instantiate TestRunner

Most unit tests for a Processor or a Controller Service start by creating an instance of the TestRunner class. In order to add the necessary classes to your Processor, you can use the Maven dependency:

```
         <dependency>
        <groupId>org.apache.nifi</groupId>
        <artifactId>nifi-mock</artifactId>
        <version>${nifi version}</version>
</dependency>
```

We create a new TestRunner by calling one of the static newTestRunner methods of the TestRunners class (located in the org.apache.nifi.util package). These methods take an argument for the Processor under test (can either be the class of the Processor to test or can be an instance of a Processor), and allow the setting of the processor name as well.

Add ControllerServices

After creating a new Test Runner, we can add any Controller Services to the Test Runner that our Processor will need in order to perform its job. We do this by calling the addControllerService method and supply both an identifier for the Controller Service and an instance of the Controller Service.

If the Controller Service needs to be configured, its properties can be set by calling the setProperty(ControllerService, PropertyDescriptor, String), setProperty(ControllerService, String, String), or setProperty(ControllerService, PropertyDescriptor, AllowableValue) method. Each of these methods returns a ValidationResult. This object can then be inspected to ensure that the property is valid by calling isValid. Annotation data can be set by calling the setAnnotationData(ControllerService, String) method.

We can now ensure that the Controller Service is valid by calling assertValid(ControllerService) - or ensure that the configured values are not valid, if testing the Controller Service itself, by calling assertNotValid(ControllerService).

Once a Controller Service has been added to the Test Runner and configured, it can now be enabled by calling the enableControllerService(ControllerService) method. If the Controller Service is not valid, this method will throw an IllegalStateException. Otherwise, the service is now ready to use.

Set Property Values

After configuring any necessary Controller Services, we need to configure our Processor. We can do this by calling the same methods as we do for Controller Services, without specifying any Controller Service. I.e., we can call setProperty(PropertyDescriptor, String), and so on. Each of the setProperty methods again returns a ValidationResult property that can be used to ensure that the property value is valid.

Similarly, we can also call assertValid() and assertNotValid() to ensure that the configuration of the Processor is valid or not, according to our expectations.

Enqueue FlowFiles

Before triggering a Processor to run, it is usually necessary to enqueue FlowFiles for the Processor to process. This can be achieved by using the enqueue methods of the TestRunner class. The enqueue method has several different overrides, and allows data to be added in the form of a byte[], InputStream, or Path. Each of these methods also supports a variation that allows a Map<String, String> to be added to support FlowFile attributes.

Additionally, there is an enqueue method that takes a var-args of FlowFile objects. This can be useful, for example, to obtain the output of a Processor and then feed this to the input of the Processor.

Run the Processor

After configuring the Controller Services and enqueuing the necessary FlowFiles, the Processor can be triggered to run by calling the run method of TestRunner. If this method is called without any arguments, it will invoke any method in the Processor with an @OnScheduled annotation, call the Processor's onTrigger method once, and then run the @OnUnscheduled and finally @OnStopped methods.

If it is desirable to run several iterations of the onTrigger method before the other @OnUnscheduled and @OnStopped life-cycle events are triggered, the run(int) method can be used to specify now many iterations of onTrigger should be called.

There are times when we want to trigger the Processor to run but not trigger the @OnUnscheduled and @OnStopped life-cycle events. This is useful, for instance, to inspect the Processor's state before these events occur. This can be achieved using the run(int, boolean) and passing false as the second argument. After doing this, though, calling the @OnScheduled life-cycle methods could cause an issue. As a result, we can now run onTrigger again without causing these events to occur by using the run(int,boolean,boolean) version of the run method and passing false as the third argument.

If it is useful to test behavior that occurs with multiple threads, this can also be achieved by calling the setThreadCount method of TestRunner. The default is 1 thread. If using multiple threads, it is important to remember that the run call of TestRunner specifies how many times the Processor should be triggered, not the number of times that the Processor should be triggered per thread. So, if the thread count is set to 2 but run(1) is called, only a single thread will be used.

Validate Output

After a Processor has finished running, a unit test will generally want to validate that the FlowFiles went where they were expected to go. This can be achieved using the TestRunners assertAllFlowFilesTransferred and assertTransferCount methods. The former method takes as arguments a Relationship and an integer to dictate how many FlowFiles should have been transferred to that Relationship. The method will fail the unit test unless this number of FlowFiles were transferred to the given Relationship or if any FlowFile was transferred to any other Relationship. The assertTransferCount method validates only that the FlowFile count was the expected number for the given Relationship.

After validating the counts, we can then obtain the actual output FlowFiles via the getFlowFilesForRelationship method. This method returns a List<MockFlowFile>. It's important to note that the type of the List is MockFlowFile, rather than the FlowFile interface. This is done because MockFlowFile comes with many methods for validating the contents.

For example, MockFlowFile has methods for asserting that FlowFile Attributes exist (assertAttributeExists), asserting that other attributes are not present (assertAttributeNotExists), or that Attributes have the correct value (assertAttributeEquals, assertAttributeNotEquals). Similar methods exist for verifying the contents of the FlowFile. The contents of a FlowFile can be compared to a byte[], and InputStream, a file, or a String. If the data is expected to be textual, the String version is preferred, as it provides a more intuitive error message if the output is not as expected.

Mocking External Resources

One of the biggest problems when testing a NiFi processor that connects to a remote resource is that we don't want to actually connect to some remote resource from a unit test. We can stand up a simple server ourselves in the unit test and configure the Processor to communicate with it, but then we have to understand and implement the server-specific specification and may not be able to properly send back error messages, etc. that we would like for testing.

Generally, the approach taken here is to have a method in the Processor that is responsible for obtaining a connection or a client to a remote resource. We generally mark this method as protected. In the unit test, instead of creating the TestRunner by calling TestRunners.newTestRunner(Class) and providing the Processor class, we instead create a subclass of the Processor in our unit test and use this:

```
         @Test
public void testConnectionFailure() {
        final TestRunner runner = TestRunners.newTestRunner(new
 MyProcessor() {
                protected Client getClient() {
                        // Return a mocked out client here.
                        return new Client() {
                                public void connect() throws IOException {
                                        throw new IOException();
                                }

                                // ...
                                // other client methods
                                // ...
                        };
                }
        });

        // rest of unit test.
}
```

This allows us to implement a Client that mocks out all of the network communications and returns the different error results that we want to test, as well as ensure that our logic is correct for handling successful calls to the client.

Additional Testing Capabilities

In addition to the above-mentioned capabilities provided by the testing framework, the TestRunner provides several convenience methods for verifying the behavior of a Processor. Methods are provided for ensuring that the Processor's Input Queue has been emptied. Unit Tests are able to obtain the ProcessContext, ProcessSessionFactory, ProvenanceReporter, and other framework-specific entities that will be used by the TestRunner. The shutdown method provides the ability to test Processor methods that are annotated to be run only on shutdown of NiFi. Annotation Data can be set for Processors that make use of Custom User Interfaces. Finally, the number of threads that should be used to run the Processor can be set via the setThreadCount(int) method.

NiFi Archives (NARs)

When software from many different organizations is all hosted within the same environment, Java ClassLoaders quickly become a concern. If multiple components have a dependency on the same library but each depends on a different version, many problems arise, typically resulting in unexpected behavior or NoClassDefFoundError errors

occurring. In order to prevent these issues from becoming problematic, NiFi introduces the notion of a NiFi Archive, or NAR.

A NAR allows several components and their dependencies to be packaged together into a single package. The NAR package is then provided ClassLoader isolation from other NAR packages. Developers should always deploy their NiFi components as NAR packages.

To achieve this, a developer creates a new Maven Artifact, which we refer to as the NAR artifact. The packaging is set to nar. The dependencies section of the POM is then created so that the NAR has a dependency on all NiFi Components that are to be included within the NAR.

In order to use a packaging of nar, we must use the nifi-nar-maven-plugin module. This is included by adding the following snippet to the NAR's pom.xml:

```
         <build>
    <plugins>
        <plugin>
            <groupId>org.apache.nifi</groupId>
            <artifactId>nifi-nar-maven-plugin</artifactId>
            <version>1.1.0</version>
            <extensions>true</extensions>
        </plugin>
    </plugins>
</build>
```

In the Apache NiFi codebase, this exists in the NiFi root POM from which all other NiFi artifacts (with the exception of the nifi-nar-maven-plugin itself) inherit, so that we do not need to include this in any of our other POM files.

The NAR is able to have one dependency that is of type nar. If more than one dependency is specified that is of type nar, then the nifi-nar-maven-plugin will error. If NAR A adds a dependency on NAR B, this will not result in NAR B packaging all of the components of NAR A. Rather, this will add a Nar-Dependency-Id element to the MANIFEST.MF file of NAR A. This will result in setting the ClassLoader of NAR B as the Parent ClassLoader of NAR A. In this case, we refer to NAR B as the Parent of NAR A.

This linkage of Parent ClassLoaders is the mechanism that NiFi uses in order to enable Controller Services to be shared across all NARs. A Controller Service must be separated into an interface that extends ControllerService and an implementation that implements that interface. Controller Services can be referenced from any Processor, regardless of which NAR it is in, as long as both the Controller Service Implementation and the Processor share the same definition of the Controller Service interface.

In order to share this same definition, both the Processor's NAR and the Controller Service Implementation's NAR must have as a Parent the Controller Service definition's NAR. An example hierarchy may look like this:

While these may seem very complex at first, after creating such a hierarchy once or twice, it becomes far less complicated. Note here that the my-controller-service-api-nar has a dependency on nifi-standard-services-api-nar. This is done so that any NAR that has a dependency on my-controller-service-api-nar will also be able to access all of the Controller Services that are provided by the nifi-standard-services-api-nar, such as the SSLContextService. In this same vane, it is not necessary to create a different "service-api" NAR for each service. Instead, it often makes sense to have a single "service-api" NAR that encapsulates the API's for many different Controller Services, as is done by the nifi-standard-services-api-nar. Generally, the API will not include extensive dependencies, and as a result, ClassLoader isolation may be less important, so lumping together many API artifacts into the same NAR is often acceptable.

Per-Instance ClassLoading

A component developer may wish to add additional resources to the component's classpath at runtime. For example, you may want to provide the location of a JDBC driver to a processor that interacts with a relational database, thus allowing the processor to work with any driver rather than trying to bundle a driver into the NAR.

This may be accomplished by declaring one or more PropertyDescriptor instances with dynamicallyModifiesClasspath set to true. For example:

```
        PropertyDescriptor EXTRA_RESOURCE = new
PropertyDescriptor.Builder()
  .name("Extra Resources")
  .description("The path to one or more resources to add to the
classpath.")
  .addValidator(StandardValidators.NON_EMPTY_VALIDATOR)
  .expressionLanguageSupported(true)
  .dynamicallyModifiesClasspath(true)
  .build();
```

When these properties are set on a component, the framework identifies all properties where dynamicallyModifiesClasspath is set to true. For each of these properties, the framework attempts to resolve filesystem resources from the value of the property. The value may be a comma-separated list of one or more directories or files, where any paths that do not exist are skipped. If the resource represents a directory, the directory is listed, and all of the files in that directory are added to the classpath individually.

Each property may impose further restrictions on the format of the value through the validators. For example, using StandardValidators.FILE_EXISTS_VALIDATOR restricts the property to accepting a single file. Using StandardValidators.NON_EMPTY_VALIDATOR allows any combination of comma-separated files or directories.

Resources are added to the instance ClassLoader by adding them to an inner ClassLoader that is always checked first. Anytime the value of these properties change, the inner ClassLoader is closed and re-created with the new resources.

NiFi provides the @RequiresInstanceClassLoading annotation to further expand and isolate the libraries available on a component's classpath. You can annotate a component with @RequiresInstanceClassLoading to indicate that the instance ClassLoader for the component requires a copy of all the resources in the component's NAR ClassLoader. When @RequiresInstanceClassLoading is not present, the instance ClassLoader simply has it's parent ClassLoader set to the NAR ClassLoader, rather than copying resources.

The @RequiresInstanceClassLoading annotation also provides an optional flag `cloneAncestorResources'. If set to true, the instance ClassLoader will include ancestor resources up to the first ClassLoader containing a controller service API referenced by the component, or up to the Jetty NAR. If set to false, or not specified, only the resources from the component's NAR will be included.

Because @RequiresInstanceClassLoading copies resources from the NAR ClassLoader for each instance of the component, use this capability judiciously. If ten instances of one component are created, all classes from the component's NAR ClassLoader are loaded into memory ten times. This could eventually increase the memory footprint significantly when enough instances of the component are created.

Additionally, there are some restrictions when using @RequiresInstanceClassLoading when using Controller Services. Processors, Reporting Tasks, and Controller Services can reference a Controller Service API in one of its Property Descriptors. An issue may arise when the Controller Service API is bundled in the same NAR with a component that references it or with the Controller Service implementation. If either of these cases are encountered and the extension requires instance classloading, the extension will be skipped and an appropriate ERROR will be logged. To address this issue, the Controller Service API should be bundled in a parent NAR. The service implementation and extensions that reference that service should depend on the Controller Service API NAR. Please refer to the Controller Service NAR Layout in the NARs section. Anytime a Controller Service API is bundled with an extension that requires it, even if @RequiresInstanceClassLoading isn't used, a WARNING will be logged to help avoid this bad practice.

Deprecating a Component

Sometimes it may be desirable to deprecate a component. Whenever this occurs the developer may use the @DeprecationNotice annotation to indicate that a component has been deprecated, allowing the developer to describe a reason for the deprecation and suggest alternative components. An example of how to do this can be found below:

```
        @DeprecationNotice(alternatives = {ListenSyslog.class}, classNames
= {"org.apache.nifi.processors.standard.ListenRELP"}, reason = "Technology
has been superseded",  )
public class ListenOldProtocol extends AbstractProcessor {
```

As you can see, the alternatives can be used to define and array of alternative Components, while classNames can be used to represent the similar content through an array of strings.