

Apache[®] NiFi[™]

for dummies[®]



Move data easily,
securely, and efficiently

Learn how to
configure processors

Set up NiFi to
regulate dataflows

- Introduction
 - Beyond the Content
- Why NiFi?
 - The Advantages to Apache Nifi
 - Nifi Core Concepts
 - NiFi Expression Language and Other Query Languages
 - JSONPath
 - XPath/XQuery
 - Getting Started with NiFi
 - Prerequisites
 - Understanding the Example Workflow
 - General Debugging & Monitoring
 - Debugging through the User Interface
 - Configuring Back-pressure
 - Checking Provenance
 - Checking the NiFi Server Logs
- NiFi Use Cases
 - Importing Datasets into a Database
- Implementing: Custom Processor
 - Prerequisite
 - Maven Archetype

Introduction

Apache NiFi was built to automate and manage the flow of data between systems and address the global enterprise dataflow issues. It provides an end-to-end platform that can collect, curate, analyze and act on data in real-time, on-premises, or in the cloud with a drag-and-drop visual interface.

Beyond the Content

NiFi is an open-source software project licensed under the Apache Software Foundation. You can find further details at <https://nifi.apache.org/>.

Why NiFi?

The information age caused a shift from an industry-based economy to a computer-based economy. For organizations, the information age has led to a situation in which immense amounts of data are stored in complete isolation, which makes sharing with others for collaboration and analysis difficult.

In response to this situation, several technologies have emerged, such as Hadoop data lakes, but they lack one major component—data movement. The capability to connect databases, file servers, Hadoop clusters, message queues, and devices to each other in a single pane is what Apache NiFi accomplishes. NiFi gives organizations a distributed, resilient platform to build their enterprise dataflows on.

In this chapter, we discuss how Apache NiFi can streamline the development process and the terminology and languages that you need to be successful with NiFi.

The Advantages to Apache Nifi

The ability to bring subject matter experts closer to the business logic code is a central concept when building a NiFi flow. Code is abstracted behind a drag-and-drop interface allowing for groups to collaborate much more effectively than looking through lines of code. The programming logic follows steps, like a white-board, with design intent being apparent with labels and easy-to-understand functions.

Apache NiFi excels when information needs to be processed through a series of incremental steps. Examples of this include:

- **Files landing on an FTP server:** An hourly data dump is made available by a vendor and needs to be parsed, enriched, and put in a database
- **Rest requests from a web application:** A website needs to make complex rest API calls and middleware must make a series of database lookups
- **Secure transmission of logs:** An appliance at a remote site needs to transmit information back to the core datacenter for analysis
- **Filtering of events data:** Event data is being streamed and needs to be evaluated for specific conditions before being archived

Nifi Core Concepts

NiFi is a processing engine that was designed to manage the flow of information in an ecosystem. Everything starts with a piece of data that flows through multiple stages of logic, transformation, and enrichment.

When building flows in NiFi, keep in mind where the data is coming from and where it will ultimately land. In many ways NiFi is a hybrid information controller and event processor. An event can be anything from a file landing in an FTP to an application making a REST request. When you consider information flow as a series of distinct events rather than a batch operation, you open a lot of possibilities.

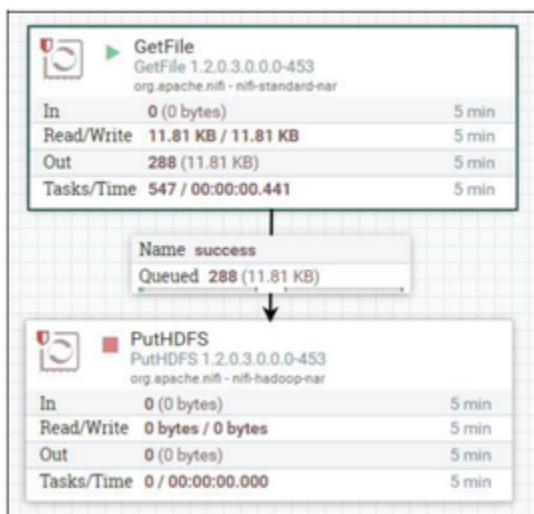
One of the biggest paradigm shifts teams may face is going from monolithic scheduled events to sequence of individual tasks. When big data first became a term, organizations would run gigantic SQL operations on millions of rows. The problem was that this type of operation could only be done after the data was fully loaded and staged. With NiFi, those same companies can consider their SQL databases as individual rows at time of ingest. This situation allows for data to be enriched and served to the end consumer in a much faster and reliable fashion. Due to the fact that each row is individually analyzed, a corrupt value would only cause that individual event to fail rather than the entire procedure.

NiFi consists of three main components:

- **Flowfiles:** Information in NiFi consists of two parts: the attributes and the payload. Flowfiles typically start with a default set of attributes that are then added to by additional operations. Attributes can be referenced via the NiFi expression language, which you can find out about in the “NiFi Expression Language and Other Query Languages” section. The payload is typically the information itself and can also be referenced by specific processors.
- **Flowfile processors:** These do all the actual work in NiFi. They’re self-contained segments of code that in most cases have inputs and outputs. One of the most common processors, GetFTP, retrieves files from an FTP server and creates a flowfile. The flowfile includes attributes about the directory it was retrieved from — such as, creation date, filename, and a payload containing the file’s contents. This flowfile can then be processed by another common processor, RouteOnAttribute. This processor looks at an incoming flowfile and applies user-defined logic based on the attributes before passing it down the chain.
- **Connections:** These detail how flowfiles should travel between processors. Common connections are for success and failure, which are simple error handling for processors. Flowfiles that are processed without fault are sent to the success queue while those with problems are sent to a failure queue. Processors such as RouteOnAttribute have custom connections based on the rules created.

Additional connection types may be Not Found or Retry and depend on the processor itself. Connections can also be Auto-Terminated if the user wishes to immediately discard a specific type of event. Configuring the advanced features of connections, such as backpressure, is covered in Chapter 3.

REMEMBER



This figure shows a basic flow incorporating these three basic concepts. A processor gets files from a local directory and creates flowfiles. These flowfiles go through the connection to another processor that puts the data into Hadoop.

Processors can be turned on and off (started/stopped), which is indicated by a green triangle (running) or red square (stopped). A stopped processor doesn't evaluate flow files.

Processors are configurable by right-click; four tabs are available:

- **Settings:** This tab allows you to rename how the processor appears and auto-terminate relationships. If a processor allows user-defined relationships to be created (such as RouteOnAttribute), they also appear here after created. More advanced settings, for example, penalty and yield duration, allow for how to handle re-trying flowfiles if the first attempt fails.

Scheduling: NiFi provides several different scheduling options for each processor. For most cases, the Timer-Driven strategy is most

appropriate. This can accommodate running on a specified interval or running as fast as NiFi can schedule it (when data is available) by setting the scheduling period to 0 seconds.

Additionally, you can increase the concurrency on this tab. Doing so allocates additional threads to the processor, but to be mindful of the number of threads available to NiFi and oversubscription.

- **Comments:** Allows developers to add comments at the per processor level.
- **Properties:** This tab is where the processor's specific settings are configured. If the processor allows custom properties to be configured, click the plus sign in the top-right to add them. Some properties allow for the NiFi Expression Language.

To tell whether a property allows for the NiFi Expression Language, hover over the question mark next to the property name and see if the Supports Expression Language property is true or false.

NiFi Expression Language and Other Query Languages

The NiFi expression language is the framework in which attributes (metadata) can be interacted with. The language is built on the attribute being referenced with a preceding `${` and proceeding `}`. For example, if you want to find the path of a file retrieved by `GetFile`, it would be `${path}`. Additional terms can be added for transformation and logic expressions, such as `contains` or `append`. Multiple variables can be nested to have a multi-variable term. Examples include

- **Check whether the file has a specific name**

```
${filename:contains('Nifi')}
```

- **Add a new directory to the path attribute**

```
${path:append('/new_directory')}
```

- **Reformat a date**

```
${string_date:toDate("yyyy-MM-DD")}
```

- **Mathematical operations**

```
${amount_owed:minus(5)}
```

- **Multi-variable greater than**

```
${variable_one:gt(${variable_two})}
```

Some processors require a Boolean expression language term to filter events such as `RouteOnAttribute` shown here.

While others, such as `UpdateAttribute`, allow more freeform use of the language.

There are two ways to use the expression language. `JSONPath` and `XPath` or `XQuery`.

JSONPath

Configure Processor

SETTINGS SCHEDULING PROPERTIES COMMENTS

Required field +

Property	Value
Routing Strategy	Route to Property name
bookWork	\$filename contains('book')
stockData	\$filename contains('stock')

\$..Version

CANCEL APPLY

\$..Version[?(@.subVersion>5)]

Configure Processor

SETTINGS SCHEDULING PROPERTIES COMMENTS

Required field +

Property	Value
Delete Attributes Expression	No value set
Store State	Do not store state
Stateful Variables Initial Value	No value set
today'sDate	\$now()
normalizedNameField	\$firstName:toUpperCase()
sanityDataBoolean	\$humidity:lt(100)

/account/user_name/first_name

ADVANCED CANCEL APPLY

When referencing JSONs with processors such as Evaluate Json Path, you use the JSONPath expression language. In this language, the JSON hierarchy is referenced with a \$ to represent the root and the names of the nested fields get a value, such as \$.account.user_name.first_name. For example, you can enumerate a list of accounts, such as \$.account[0].user_name.first_name.

- Search any level of JSON for a field called Version

- Filter only for subversions greater than 5

XPath/XQuery

The XPath/Query language is available for accessing data in XMLs through processors such as EvaluateXPath and EvaluateXQuery.

Much like JSONPath, it allows for data to either be exactly specified or searched:

- Specify the value of the account holder's first name

- Specify the value of the first account if multiple accounts are present in the XML

/account[0]/user_name/first_name

- Search any level of XML for a field called Version

//Version

- Filter only for subversions greater than 5

//Version[subVersion>5]

Getting Started with NiFi

Apache NiFi is one of the most flexible, intuitive, feature rich dataflow management tools within the open-source community. NiFi has a simple drag-and-drop user interface- (UI), which allows administrators to create visual -dataflows and manipulate the flows in real time and it provides the user with information pertaining to audit, lineage, and backpressure.

For example, to really begin to understand some of the capabilities, it's best to start with a simple Hello World dataflow. The traditional Hello World example (as every technologist is used to starting with when learning any programming language) is a bit different with a dataflow management tool such as NiFi. This simple example demonstrates the flexibilities, ease of use, and intuitive nature of NiFi. In this chapter, we explain how to import a NiFi template, create a NiFi dataflow, and how data is processed and stored.

Prerequisites

- ☐ Install Nifi in Docker Container or Install Locally
- ☐ A latest browser
- ☐ (optional) Favourite IDE for Custom processor/ Custom controller service

▼ [docker-compose.yml](#)

```
# contributor license agreements. See the NOTICE file distributed
# with
# this work for additional information regarding copyright
# ownership.
# The ASF licenses this file to You under the Apache License,
# Version 2.0
# (the "License"); you may not use this file except in compliance
# with
# the License. You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions
# and
# limitations under the License.

version: "3"
services:
  zookeeper:
    hostname: zookeeper
    container_name: zookeeper
    image: 'bitnami/zookeeper:latest'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  nifi:
    image: apache/nifi:1.11.1
    hostname: nifihost
    container_name: nifi
    ports:
      - 8080:8080 # Unsecured HTTP Web Port
      - 6688:6688 # exposing the ports to outside docker container
  availability
    environment:
      - NIFI_WEB_HTTP_PORT=8080
```

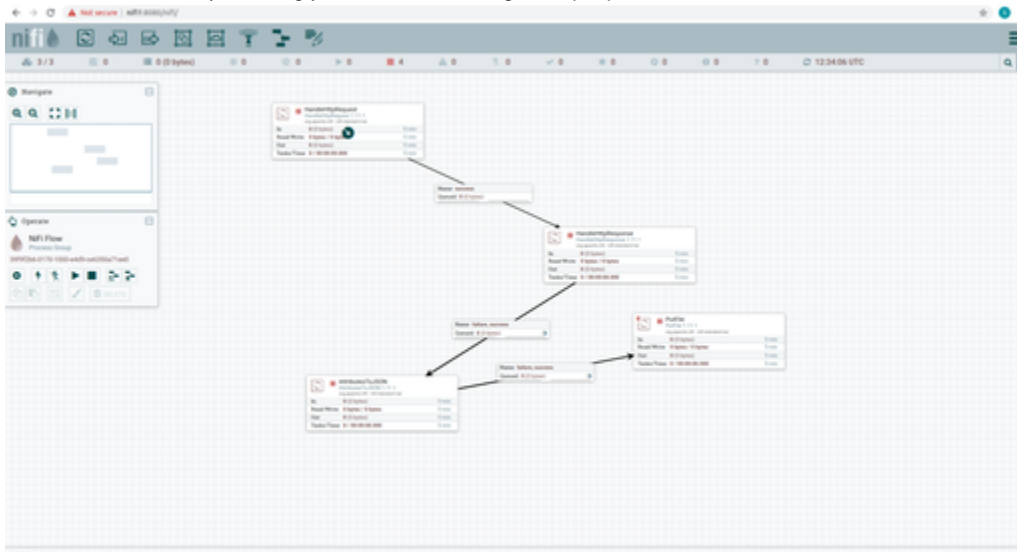
```

- NIFI_CLUSTER_IS_NODE=true
- NIFI_CLUSTER_NODE_PROTOCOL_PORT=8082
- NIFI_ZK_CONNECT_STRING=zookeeper:2181
- NIFI_ELECTION_MAX_WAIT=1 min
volumes:
- nifi_conf:/opt/nifi/nifi-current/conf #volume helps to keep
data like properties even after docker image is umounted or container
removed.

volumes:
  nifi_conf:
    driver_opts:
      type: none
      device: ${PWD}/nificonf
      o: bind

```

When Nifi Successfully Running you can start creating example processors inter communication



This is a simple Http Server processor which response 200 status code for any request. And a Flow file gets generated in server machine/container for every success or failure response.

To know more how to use the UI and create the workflow, visit here : [UI Interface Guide](#)

Understanding the Example Workflow

The **HandleHttpRequest** NiFi processor starts a HTTP server in NiFi and listens for HTTP requests on a specific port. In this case, the NiFi processor is already set up on port 6688. When you pointed your browser to

<http://localhost:6688> NiFi handled this request and passed it to the next process.

The next process which is **HandleHttpResponse** NiFi processor will start processing the based on `StandardHttpContextMap`. The processor have the configuration for Http Response Status code and content. Ideally status code should set to 200. To any RestClient, response will be visible.

General Debugging & Monitoring

In this chapter, we discuss how to interpret information about your processes through the user interface, set up backpressure to allow NiFi to regulate itself, use provenance to help with debug-ging efforts when things don't go as planned, and monitoring processes through the NiFi server logs when you want more detail.

Debugging through the User Interface

You can glean a lot of information right from the user interface, through the status bar, the Summary window, and the Status History menu.

Status bar

The status bar is located at the top of the user interface under the drag-and-drop toolbox. It provides metrics related to:

- Nodes in the cluster
- Threads running

- Flowfile count and content size
- Remote process groups in transmitting or disabled state
- Processors status (for example, which ones are running, stopped, invalid, disabled, the last time the UI was refreshed)

The amount of information reported in the status bar is minimal. When you need more in-depth information, choose the Summary found in the menu.

Summary

The Summary window contains tabs for processors, input ports, output ports, remote process groups (RPGs), connections, and process groups.

The processor groups located on the canvas contain their own status bars and general metrics; they're also available on the Summary's Process Groups tab in a tabular report. The **Connections** tab provides basic information as well: name, relation type being connected, the destination, queue size, % of queue threshold used, and output size in bytes. Metrics that track the input and outputs are tracked over a five-minute window. With the Summary's Connections tab, all the funnels on the canvas can be identified to validate that they're being used downstream and not just dead ends from other processors

NiFi Summary

PROCESSORS INPUT PORTS OUTPUT PORTS REMOTE PROCESS GROUPS **CONNECTIONS** PROCESS GROUPS

Displaying 2 of 4

Funnel by destination ▼

Source Name	Name	Destination Name	In / Size 5 min	Queue / Size	Queue / Size Threshold	Out / Size 5 min
HashContent	failure, success	Funnel	0 (0 bytes)	6,961 (0 bytes)	70% / 0%	0 (0 bytes)
HashContent	failure, success	Funnel	0 (0 bytes)	0 (0 bytes)	0% / 0%	0 (0 bytes)

Last updated: 17:41:16 EDT

system diagnostics

Additionally, at the bottom right of the Summary on any tab is the system diagnostics link. There are three tabs:

- **JVM** tab shows metrics about on and off-heap utilization and garbage collection counts along with total time.
- **System** as shown in figure below shows the number of CPU cores and the amount of space used on the partition that the repository is stored on.
- **Version** contains detailed build numbers of NiFi, the version of Java NiFi is running with, and details on the OS.

NiFi Summary

PROCESSORS INPUT PORTS OUTPUT PORTS REMOTE PROCESS GROUPS CONNECTIONS **PROCESS GROUPS**

Displaying 3 of 3

Filter

Name	Value	Time 5 min
GenerateFlowFile		
HashContent		
HashContent		

System Diagnostics

JVM SYSTEM **VERSION**

17:38:14 EDT

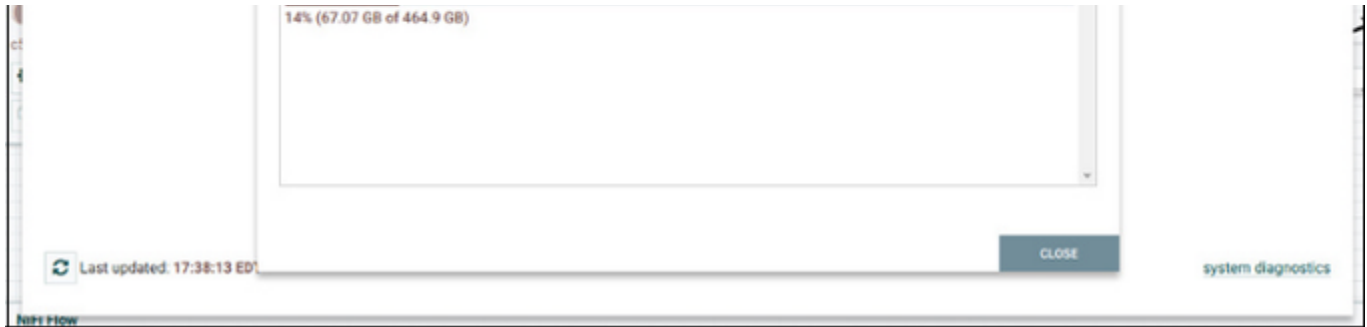
Available Processors: 8 Processor Load Average: 2.70

FlowFile Repository Storage

Usage: 14% (67.07 GB of 464.9 GB)

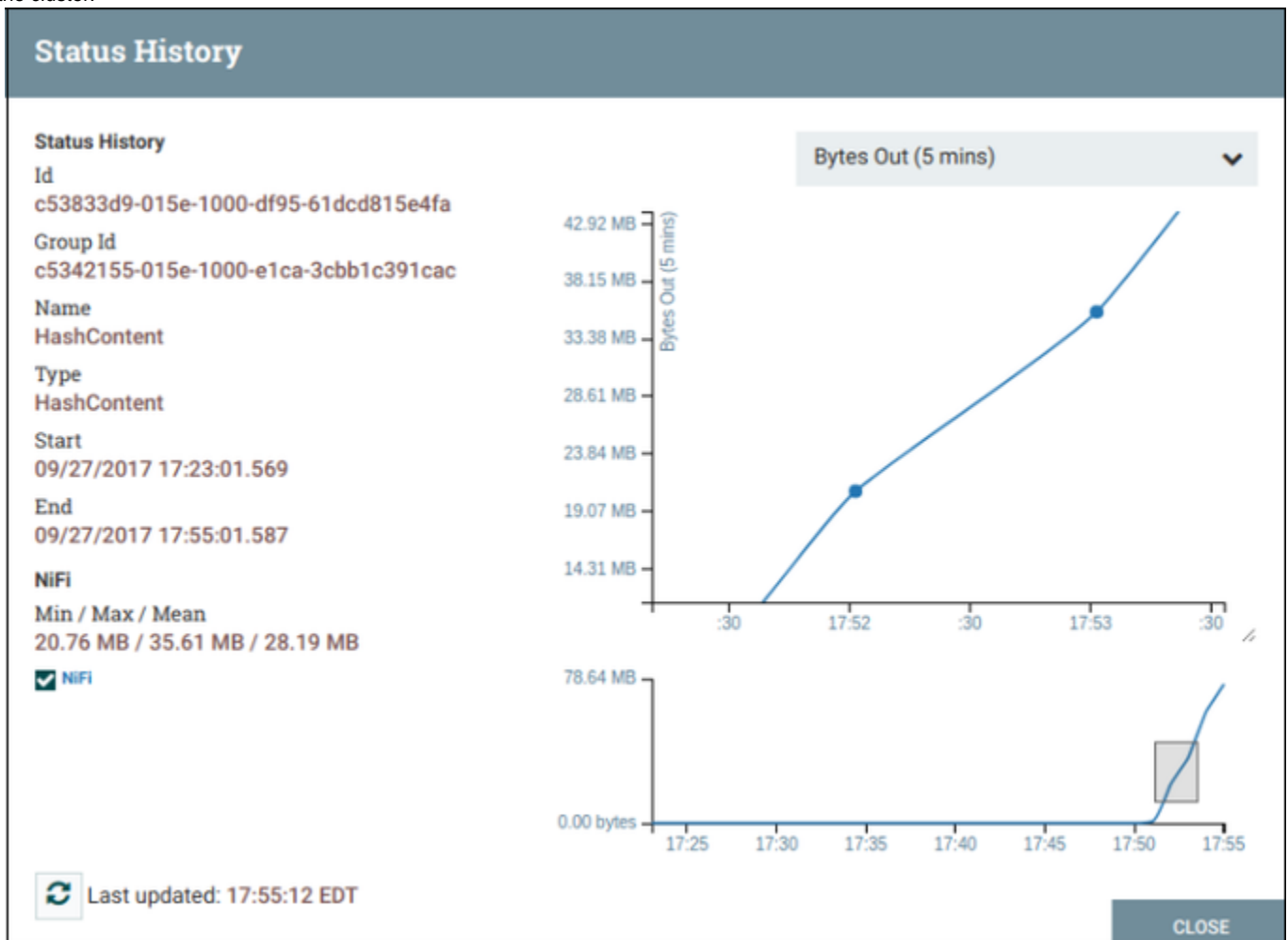
Content Repository Storage

Usage For Default:



Status History

The Status History menu is one of the most useful features, next to Data Provenance, in debugging a slow flow. The Status History menu contains all the generic information expected such as the name and the time the status has been collected for. The graph in the menu, as shown below, can visualize many metrics related to the processor or connection (over 5 minutes) and includes separate plotting for each NiFi node in the cluster.




The line graph in the Status History menu can be zoomed in by dragging from one part of the graph to the next.

Configuring Back-pressure

Back-pressure isn't typically thought of as a first-class citizen in many data movement systems, but NiFi provides it as a first-class feature. From an operational perspective, back-pressure enables you to design a system that self regulates the amount of storage it's utilizing to prevent it from crashing! In NiFi, backpressure is configured at the connection level, where you can manage the backpressure policies.

It's important to understand how the backpressure is configured in the flow to understand its behavior. If a connection is ever completely utilized by storage or flowfile count, the processor upstream of the connection stops processing and waits for room to be made in the connection. This problem is typically caused by a slow processor downstream of the connection that can't consume the flowfiles as fast as upstream processors can place them into the connection queue.

The simplest visual to monitor appears on the Connections tab of the Summary window with both flowfile and storage size icons that represent capacity: Green (most), Yellow, and Red (least). (See the earlier "Summary" section for more about the Connections tab.) It also lists the source and target processor names along with utilization and last five-minute metrics. The Connections tab can be targeted for a specific node or the entire NiFi cluster.

 Use this information to identify specific ingestion methods that skew, such as a Kafka processor that has a high skew to a specific keyed partition where a specific NiFi node would be responsible for a single partition receiver.

You can configure back-pressure from two perspectives: value and infrastructure storage:

- **Value:** Accepting that it's impossible to store everything is the first step to designing systems that have the capability to self regulate their contents.
- **Storage:** The infrastructure itself has physical limitations on total storage available for Flowfile, Content, and Provenance repositories to use. By defining the value of specific messages in the flow, the flowfiles of lesser importance can be dropped while holding onto more important ones.


NiFi supports three ways to configure the backpressure policies of a connection:

- **Flowfile count:** Ensure only a specific number of files are in the queue to be processed and expire others.
- **ContentSize:** Limit the amount of downstream flow storage used and also ensure that the total storage for all connections is set up in a manner that prevents the flow from filling a disk completely.
- **Time:** Expire data that remained in the queue for too long so that it no longer holds any value in being processed.

Filling the storage mounts on a NiFi server can lead to very odd behavior of the repositories, which can result in requiring special actions to restore normal operation of the NiFi server that filled. Refer to the Summary menu's system diagnostics link for detailed storage use by the NiFi nodes (see the earlier section).

The Flowfile repository is much smaller than the Content repository. For example, a sample flow in a NiFi cluster with three nodes holding 32,500 flowfiles totaling 872.5MB results in the repositories on a single node (1/3) looking like the following table. It's important to note that in a cluster, work is distributed and some servers could have more work depending on the ingestion and transformations taking place in the flow on each server.

Node	Repository	Size
1	Content	285MB
1	Flowfile	5.4MB
2	Content	305MB
2	Flowfile	6.2MB
3	Content	265MB
3	Flowfile	5.9MB

 If you're trying to empty a queue and flowfiles remain, the downstream processor may have a lease on the files. Stopping the processor allows you to clear out these files.

Checking Provenance

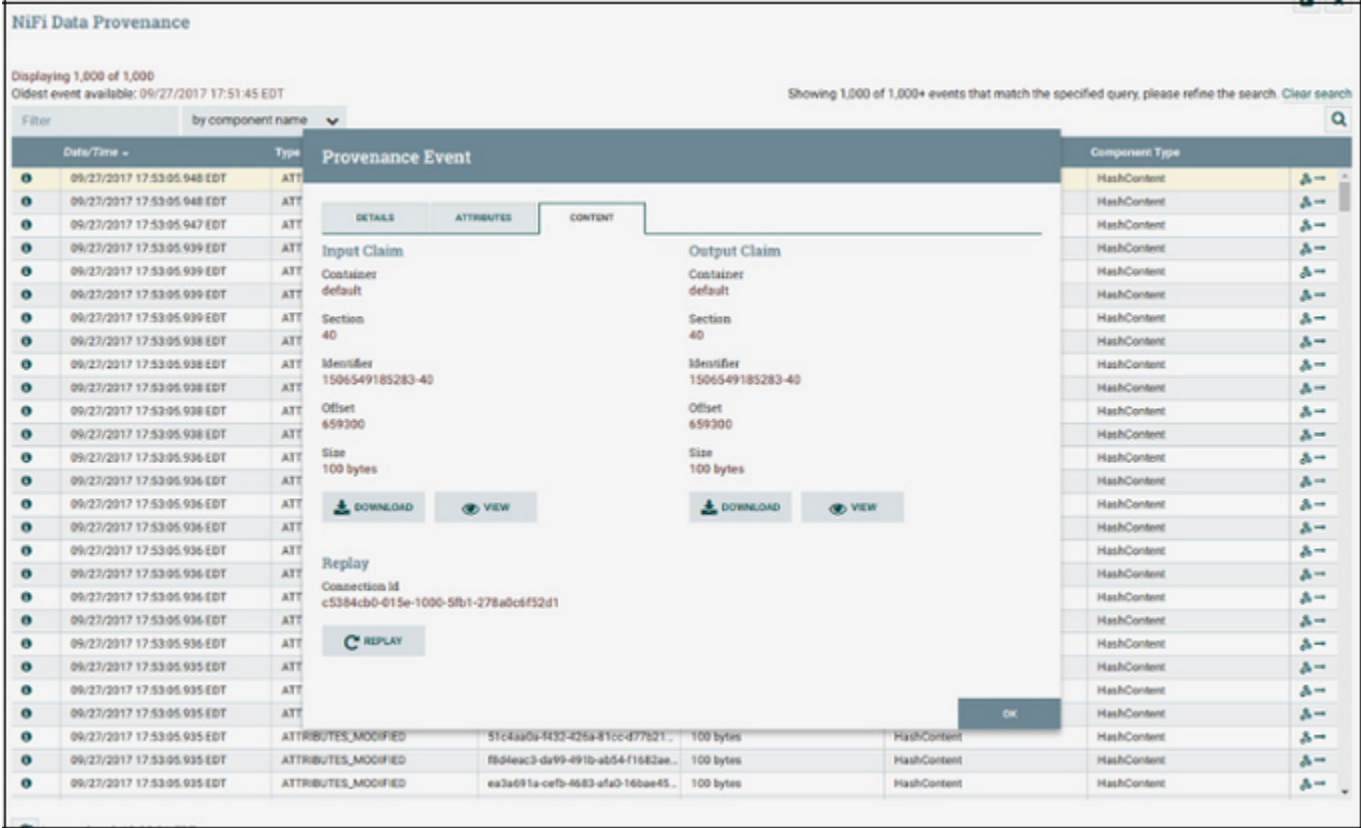
NiFi's data provenance provides debugging capabilities that allow for flowfiles themselves to be tracked from start to end inside the workflow. Flowfiles can have their contents inspected, downloaded, and even replayed. The combination of these features enables ease of troubleshooting to find out why a specific path was taken in the workflow. This capability allows you to make minor changes to the workflow based on what occurred and replay the message to ensure the new path is correctly taken.

Both processors and connections have data provenance available by right-clicking; alternatively, you can access the complete Provenance repository from the Provenance menu. The Provenance menu includes the Date/Time, ActionType, the Unique Flowfile ID, and other stats. On the far left is a small "i" encircled in blue; click this icon, and you get the flowfile details. On the right, what looks like three little circles connected together is Lineage.

Lineage is visualized as a large directed acyclic graph (DAG) that shows the steps in the flow where modifications or routing took place on the flowfile. Right-click a step in the Lineage to view details about the flowfile at that step or expand the flow to understand where it was potentially

cloned from. At the very bottom left of the Lineage UI is a slider with a play button to play the processing flow (with scaled time) and understand where the flowfile spent the most time or at which point it got routed.

Inside the flowfile details, you can find a detailed analysis of both the content and its metadata attributes. More interesting metrics are potentially the Queue Positions and Durations along with the worker the flowfile is located on. The Content tab allows you to investigate before and after versions of a flowfile after it's been processed (see figure below); just read the data in the browser or download it for later. To correct a problem, use the Replay capability to make a connection to the flow and replay the flowfile again. (And then inspect it again to be sure it runs the way you want.)



Checking the NiFi Server Logs

Each NiFi server has a set of application and bootstrapping logs. NiFi uses SLF4J to provide a robust and configurable logging framework that you can configure to provide as much detail as you want. The logs contain detailed information about processes occurring on the server.

By default, the NiFi server logs are located in the `logs/` directory found in the NiFi folder. This folder is also called `$NIFI_HOME`. If you downloaded and untarred/unzipped NiFi, the directory is `NIFI_HOME`.

The `nifi-app.log` application log contains more details about processors, remote process groups (for site to site), Write Ahead Log functions, and other system processes.

The `nifi-bootstrap.log` bootstrap contains entries on whether the NiFi server is started, stopped, or dead. It also contains the complete command with classpath entries used to start the NiFi service.

💡 The `logback.xml` located in `$NIFI_HOME/conf` can be edited on the fly without having to restart NiFi. It takes approximately 30 seconds before the new logging configuration takes effect. The log level can be configured per node and isn't a clusterwide configuration. For example, to only change the Processor log level, edit the `logback.xml` and change the logger line for `org.apache.nifi.processors` to `INFO` from `WARN`.

NiFi Use Cases

Planning the first NiFi data integration project requires attention to:

- **Data volume and velocity:** Pulling flat files from a -monitoreddirectory often leads to large files made available infrequently, which can require lots of memory to process. In contrast, when data is pushed from an external source to a NiFi listener over a TCP/IP port, each data row tends to be small in size but is received frequently.
- **Data types to ingest:** NiFi has the ability to support both binary and text data sets.
- **Capacity of connected systems:** When considering system capacity needed, pay close attention to each of the connected- systems' capabilities to accept the data as it becomes avail-able, support temporary content data storage, and store data long term.

While NiFi can support many different ingest use cases, in this chapter we examine three sample use case scenarios.

Importing Datasets into a Database

In this scenario, the requirement is to monitor a directory on disk and on a scheduled basis read all the files in the directory, validate the data in those files, and finally write the valid records into a database.

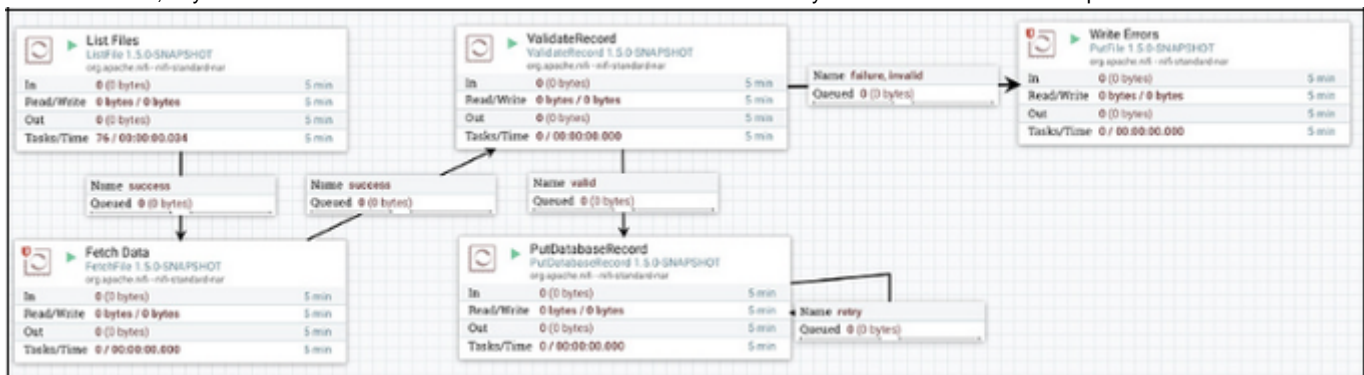
The flow consists of the following steps:

1. Periodically scans the directory and fetches the contents whenever a new file arrives.
 - The ListFile and FetchFile processors accomplish this step.
2. Validates the contents of the data.
 - The ValidateRecord processor, which accomplishes this step, is configured with a schema that describes how the data should look. In this case, the flow routes invalid records to a processor and writes them to an errors directory. The ValidateRecord processor is configured with a Record Reader so that it's capable of processing any kind of record-oriented data, such as CSV, JSON, Avro, or even unstructured log data.
3. Publishes the data to the database using the PutDatabaseRecord processor.
 - Again, this processor uses a Record Reader so that it can process any record-oriented data.

Failures during data integration can happen, so you need to plan for errors. NiFi handles this nicely through the ability to route exceptions into a failure queue to either hold for reprocessing or to write to an output file for offline editing.

The sample scenario validates that the data file contents match a schema and any invalid records are written to an error directory via the PutFile processor. This processor can also be stopped to hold the data in the queue within NiFi. From here, you can review the input content, make any necessary corrections to the flow, and reprocess the data. See Figure below

At the same time, any records that did match the schema are written to the database by the PutDatabaseRecord processor.



This processor allows you to configure where the data should be published and how, including mapping of field names in the data to database column names.

The concepts used for this NiFi dataflow can be extended to a variety of different import scenarios. Eg. pulling from a relational database or a RESTful query.

Implementing: Custom Processor

Prerequisite

- ☐ Maven installation required. how to install maven
- ☐ Your Favourite IDE eclipse, IntelliJ

Maven ArcheType

We can also use this command to generate maven archetype:

```
$ mvn archetype:generate -DarchetypeGroupId=org.apache.nifi -  
DarchetypeArtifactId=nifi-processor-bundle-archetype -  
DarchetypeVersion=1.11.1 -DnifiVersion=1.11.1
```

After generation the folder structure would look like below

```
nifiProjects/processor  
├── nifi-custom-nar  
│   └── pom.xml  
├── nifi-custom-processors  
│   ├── pom.xml  
│   └── src  
│       ├── main  
│       │   ├── java  
│       │   │   ├── com  
│       │   │   │   ├── custom  
│       │   │   │   │   ├── processors  
│       │   │   │   │   │   └── MyProcessor.java  
│       │   └── resources  
│       │       ├── META-INF  
│       │       └── services  
│       │           └── org.apache.nifi.processor.Processor  
│       └── test  
│           ├── java  
│           │   ├── com  
│           │   │   ├── custom  
│           │   │   │   ├── processors  
│           │   │   │   │   └── MyProcessorTest.java  
│           └── resources  
│               └── test.json  
├── pom.xml  
└── target  
    ├── maven-shared-archive-resources  
    │   └── META-INF  
    │       ├── DEPENDENCIES  
    │       ├── LICENSE  
    │       └── NOTICE
```

To start working/implementation on nifi custom processor import project `nifi-custom-processor` into your favourite IDE.

Open the `MyProcessor.java` file to implement your custom logic.

```
/*  
 * Licensed to the Apache Software Foundation (ASF) under one or more
```

```
* contributor license agreements. See the NOTICE file distributed with
* this work for additional information regarding copyright ownership.
* The ASF licenses this file to You under the Apache License, Version
2.0
* (the "License"); you may not use this file except in compliance with
* the License. You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
```

```
package com.custom.processors;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.atomic.AtomicReference;

import org.apache.commons.codec.Charsets;
import org.apache.commons.io.IOUtils;
import org.apache.nifi.annotation.behavior.ReadsAttribute;
import org.apache.nifi.annotation.behavior.ReadsAttributes;
import org.apache.nifi.annotation.behavior.WritesAttribute;
import org.apache.nifi.annotation.behavior.WritesAttributes;
import org.apache.nifi.annotation.documentation.CapabilityDescription;
import org.apache.nifi.annotation.documentation.SeeAlso;
import org.apache.nifi.annotation.documentation.Tags;
import org.apache.nifi.annotation.lifecycle.OnScheduled;
import org.apache.nifi.components.PropertyDescriptor;
import org.apache.nifi.flowfile.FlowFile;
import org.apache.nifi.processor.AbstractProcessor;
import org.apache.nifi.processor.ProcessContext;
import org.apache.nifi.processor.ProcessSession;
import org.apache.nifi.processor.ProcessorInitializationContext;
import org.apache.nifi.processor.Relationship;
import org.apache.nifi.processor.exception.ProcessException;

import com.fasterxml.jackson.databind.ObjectMapper;

@Tags({"example"})
@CapabilityDescription("Provide a description")
@SeeAlso({})
@ReadsAttributes({@ReadsAttribute(attribute="", description="")})
```

```

@WritesAttributes({@WritesAttribute(attribute="", description="")})
public class MyProcessor extends AbstractProcessor {

    //In procssor settings these properites will appear there
    public static final PropertyDescriptor MY_PROPERTY = new
PropertyDescriptor
        .Builder().name("MY_PROPERTY")
        .displayName("My property")
        .description("Example Property")
        .build();

    //The relation based on which the next processor will be linked.
    eg. SUCCESS/FAILURE
    public static final Relationship SUCCESS = new Relationship.
Builder()
        .name("Sucess")
        .description("This is for success")
        .build();

    private List<PropertyDescriptor> descriptors;

    private Set<Relationship> relationships;

    @Override
    protected void init(final ProcessorInitializationContext context) {
        final List<PropertyDescriptor> descriptors = new
ArrayList<PropertyDescriptor>();
        descriptors.add(MY_PROPERTY);
        this.descriptors = Collections.unmodifiableList(descriptors);

        final Set<Relationship> relationships = new
HashSet<Relationship>();
        relationships.add(SUCCESS);
        this.relationships = Collections.unmodifiableSet(relationships);
    }

    @Override
    public Set<Relationship> getRelationships() {
        return this.relationships;
    }

    @Override
    public final List<PropertyDescriptor>
getSupportedPropertyDescriptors() {
        return descriptors;
    }

    @OnScheduled
    public void onScheduled(final ProcessContext context) {

```

```

    }

    //Put you custom logic here.
    @Override
    public void onTrigger(final ProcessContext context, final
ProcessSession session) throws ProcessException {
        FlowFile flowFile = session.get();
        if ( flowFile == null ) {
            return;
        }
        final AtomicReference<String> value = new AtomicReference<>();
        //To read the flowfile content.
        session.read(flowFile, in -> {
            try{
                String json = IOUtils.toString(in,Charsets.UTF_8);
                String result = new ObjectMapper().readTree(json).get
("hello").textValue();
                value.set(result);
            }catch(Exception ex){
                ex.printStackTrace();
                getLogger().error("Failed to read json string.");
            }
        });

        // Write the results to an attribute, the write method is used
to write data to flow
        String results = value.get();
        if(results != null && !results.isEmpty()){
            flowFile = session.putAttribute(flowFile, "match", results);
        }

        // To write the results back out ot flow file
        flowFile = session.write(flowFile, out -> out.write(value.get().
getBytes()));

        //Finally every flow file that is generated needs to be deleted
or transfered.
        session.transfer(flowFile, SUCCESS);
    }
}

```

To Test your Processor using Unit Test

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements. See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version
2.0

```



```

* (the "License"); you may not use this file except in compliance with
* the License. You may obtain a copy of the License at
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
package com.custom.processors;

import static org.junit.Assert.assertEquals;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

import org.apache.nifi.util.TestRunner;
import org.apache.nifi.util.TestRunners;
import org.junit.Before;
import org.junit.Test;

public class MyProcessorTest {

    private TestRunner testRunner;
    private static String testJsonPath = "src/test/resources/test.json";
    @Before
    public void init() {
        testRunner = TestRunners.newTestRunner(MyProcessor.class);
    }

    @Test
    public void testProcessor() throws IOException {
        String content = new String(Files.readAllBytes(Paths.get(
(testJsonPath)), StandardCharsets.UTF_8));
        testRunner.enqueue(content);
        testRunner.run();
        testRunner.assertQueueEmpty();
        testRunner.assertTransferCount(MyProcessor.SUCCESS, 1);
        String outputToAssert = new String(testRunner.
getFlowFilesForRelationship(MyProcessor.SUCCESS).get(0).toByteArray());
        assertEquals("The content should be equal", "Hello world",
outputToAssert);
        testRunner.shutdown();
    }
}

```

```
}  
  
}
```

To deploy `MyProcessor` in `nifi`, we will change our directory to our processor project directory and build the project:

```
$ cd <processor folder>  
$ mvn clean install
```

Now, copy the build `nifi-custom-nar-1.0-SNAPSHOT.nar` file to `nifi lib` directory and restart `nifi`:

```
$ cp nifi-custom-nar/target/nifi-custom-nar-1.0-SNAPSHOT.nar NIFI_HOME  
/lib  
$ NIFI_HOME/bin/nifi.sh start
```

You can also copy the `*.nar` file to `$NIFI_HOME/bin/extensions`

Manage your data-in-motion with Apache NiFi

Apache NiFi is an easy to use, powerful, and reliable system to process and distribute data. It provides an end-to-end platform that can collect, curate, analyze, and act on data in real-time, on-premises, or in the cloud with a drag-and-drop visual interface. This book offers you an overview of NiFi along with common use cases to help you get started, debug, and manage your own dataflows.

Inside...

- Discover the advantages of

Apache NiFi

- Create a NiFi dataflow
- Troubleshoot processors
- Learn about three NiFi use cases
- Accelerate data replication

**for
dummies®**