Brought to you by:

CLOUDERA

Apache® NiFi™

for dummies®

A Wiley Brand

Move data easily, securely, and efficiently

Learn how to configure processors

Set up NiFi to regulate dataflows

Cloudera 2nd Special Edition

Foreword by Mark Payne

Cloudera

At Cloudera, we believe data can make what is impossible today, possible tomorrow. Cloudera taught the world the value of data, creating an industry and ecosystem powered by the relentless innovation of the open source community. We empower our customers, leaders in their industries, to quickly and easily transform complex data into clear and actionable insights. Through our hybrid data platform, organizations are able to build their data-driven future by getting data— no matter where it resides—into the hands of those that need it. Learn more at cloudera.com.

Connect with Cloudera

About Cloudera: cloudera.com/more/about.html

Read our blog: https://blog.cloudera.com/

Follow us on Twitter: twitter.com/cloudera

Visit us on Facebook: facebook.com/cloudera

Join the Cloudera Community: community.cloudera.com

Read about our customers' successes: cloudera.com/more/customers.html.

Apache® NiFi™

for dummies®
A Wiley Brand

Apache® NiFi™

Cloudera 2nd Special Edition

By Guy Livini, Hima Lanka, James Herron, John Kuchmek, Richard Walden, Steven Matison, & Steven Seguna

FOREWORD BY **Mark Payne**

for dummies®
A Wiley Brand

Apache® NiFi™ For Dummies®, Cloudera 2nd Special Edition

Published by: John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

ISBN 978-1-119-81055-1 (pbk); ISBN 978-1-119-81056-8 (ebk)

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Development Editor: Rebecca Senninger

Acquisition Editor: Ashley Coffey

Editorial Manager: Rev Mengle

Business Development Representative: Molly Daugherty

Production Editor: Tamilmani Varadharaj

Special Thanks: Vinicius Cardoso, Hanneh Bandi, Ganapathy Raman

Table of Contents

Foreword

Nearly ten years ago, I was presented with an amazing opportunity. I was fortunate enough to join a team of three incredibly talented engineers to build a new platform. This platform would be responsible for handling the ever-increasing volumes of data that would be streamed through my organization. It would have to allow users to quickly add new sources of data on the fly and route, analyze, process, and transform the data, all before delivering it to its final destination. In short, the goal was to get the right data to the right place, in the right format and schema, at the right time.

Thus began a long and engaging journey. Over the next year, I would take on more and more responsibilities, ultimately taking full responsibility for the development of the framework. The volume and variety of the data continued to increase. My organization decided that if the software were to be open sourced, it would not only benefit us but also many others who were on a similar journey. So, in November of 2014, this software was donated to the Apache Software Foundation and became known as Apache NiFi.

Since its debut, NiFi has been adopted by companies and organizations across every industry. The authors of this book have been with them through it all — training the users, assessing the strengths and weaknesses of the platform, and even getting their hands dirty to improve the code. They have seen a vast number of different use cases for NiFi across these industries and been active in the day-to-day use of the software to solve their critical dataflow problems. I have had the pleasure of working alongside some of the authors to tackle their most difficult dataflow challenges and learn from their experiences. Others have written of their experiences using NiFi and established that they have a great understanding of not just the software itself but also where it came from and where it is heading.

This book is not intended to provide an in-depth understanding of every aspect of NiFi but rather is meant to provide an understanding of what NiFi is and explain when, how, and why to use NiFi. Additionally, it will explore the features that make the software unique. Through tutorials, examples, and explanations, it provides an excellent overview and walkthrough of NiFi that will benefit the uninitiated and experienced users alike.

While reading this book, you will gain a firm grasp on NiFi fundamentals and how to use the software. You should also be able to relate them to some of the challenges that you are facing and understand how NiFi can help to address them. Most importantly, I hope that you enjoy the read and that it encourages you to read more about NiFi and explore it on your own.

Cheers,

Mark Payne

Technical Lead, Apache NiFi, Cloudera

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.

About This Book

This book gives you an overview of NiFi, why it's useful, and some common use cases with technical information to help you get started, debug, and manage your own dataflows.

Icons Used in This Book

REMEMBER

Remember icons mark the information that's especially important to know.

TIP

The Tip icon points out helpful suggestions and useful nuggets of information.

WARNING

The Warning icon marks important information that may save you headaches.

Beyond the Book

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

Where to Go from Here

The book is modular, so you can start with Chapter 1, but feel free to roam around to the chapters that fit best. You can also pick out a topic that interests you from the Table of Contents.

IN THIS CHAPTER

- What to use NiFi for
- Learning NiFi core concepts
- Understanding the expression and query languages for NiFi

Chapter 1
Why NiFi?

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

Several technologies have emerged in response to this situation, such as data lakes, but they lack one major component — data movement. The capability to connect databases, file servers, Hadoop clusters, message queues, and devices 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 you need to be successful with NiFi.

The Advantages of 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 groups to collaborate much more effectively than looking through

lines of code. The programming logic follows steps, similar to a whiteboard, with design intent 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 data center 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 designed to manage a continuous flow of information as a series of events in an ecosystem, as opposed to batch operations that need to wait for a full dataset to be loaded. Everything starts with a piece of data that flows continuously through multiple stages of logic, transformation, and enrichment.

REMEMBER

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 many possibilities.

One of the biggest paradigm shifts teams may face is going from monolithic scheduled events to a 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 the time of ingestion. This situation allows for data to be enriched and served to the end consumer faster and with more reliability. Because 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 the information content itself, the data, and can also be referenced by specific processors.
- **Processors:** Do all the actual work in NiFi. They are 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 the 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 to determine which route the data should flow through 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.

Figure 1-1 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 or another filesystem.

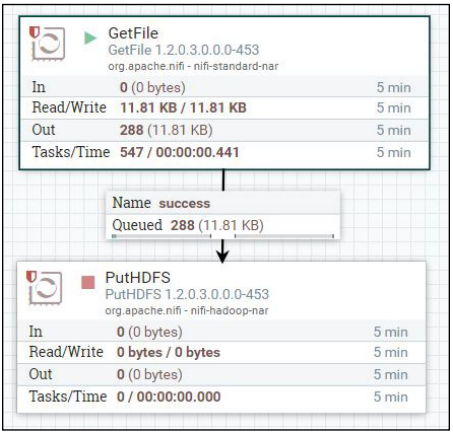

FIGURE 1-1: The first flow.

REMEMBER

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 flowfiles.

Processors are configurable by righting-click and choosing Configure; five tabs are available:

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

REMEMBER

Additionally, you can increase the concurrency on this tab. Doing so allocates additional threads to the processor, but 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.

TIP

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

NiFi Expression Language and Other Query Languages

The NiFi Expression Language is the framework with 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})}
```


TIP

When writing an expression language statement, enter **${** and then press Ctrl+Space to get a list of possible functions you can choose from to auto-complete your statement. There is also documentation available where you can look up a function.

Some processors require a Boolean expression language term to filter events such as `RouteOnAttribute`, shown in Figure 1-2.

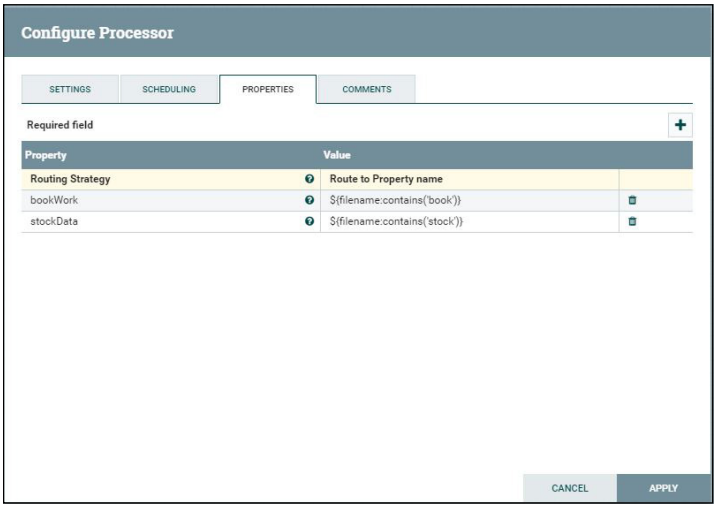

FIGURE 1-2: This processor requires a Boolean expression.

While others, such as `UpdateAttribute`, allow more freeform use of the language, as shown in Figure 1-3.

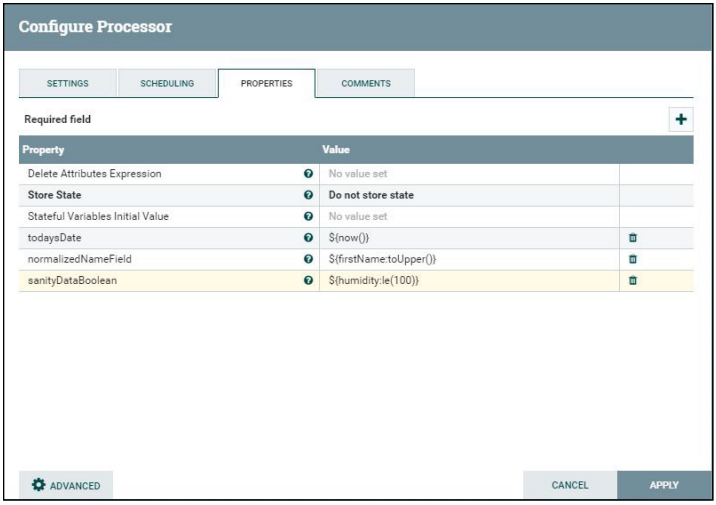

FIGURE 1-3: This processor gives you a lot of freedom in your language.

JSONPath

When referencing JSONs with processors such as `Evaluate JsonPath`, 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`. Additional complex operations are available:

- **Search any level of JSON for a field called Version**

```
$..Version
```

- **Filter only for subversions greater than 5**

```
$..Version[?(@.subVersion>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**

```
/account/user_name/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]
```

IN THIS CHAPTER

- Importing a NiFi template
- Creating a NiFi dataflow
- Understanding the Hello World example

Chapter 2
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 flexibility, 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.

DOWNLOADING NIFI AND CLONING A REPO

Installing and starting NiFi is outside the scope of this book. Information on how to install and start NiFi can be found at `https://nifi.apache.org/docs.html`.

You can download Apache NiFi at `https://nifi.apache.org/download.html`. The demo GitHub repository we follow throughout this book was tested with version 1.16.0 but should work with later versions of NiFi. Once NiFi is running, you need to clone or download the GitHub repository to follow along with the Hello World dataflow example in this chapter.

By default, NiFi generates a username and password on startup to log into the NiFi UI. If you want to set the username and password, you can run the following command from the directory that you installed NiFi into:

```
./bin/nifi.sh set-single-user-credentials <user
   name> <password>
```

Note: Because NiFi out of the box is secured, you need to set a password that is, at least, 12 characters long.

If you would rather use the generated credentials to log into the NiFi UI, you can navigate to the logs directory where you had installed NiFi and search the `nifi-app.log` file for *Generated Username* and *Generated Password*. You need to exclude the brackets `[]`. For example, if you search the log file and find `Generated Username [5c398133-cd42-49b1-8e16-afd53613d015]`, then your username is 5c398133-cd42-49b1-8e16-afd53613d015.

To clone the GitHub repository from a command line, type the following command:

```
git clone https://github.com/drnice/NifiHelloWorld.git
```

To download the GitHub repository, point your web browser at `https://github.com/drnice/NifiHelloWorld`, click the green Clone or Download button, and click Download ZIP. After it's downloaded, extract the zip file.

After the repository is cloned or downloaded and extracted, note the location of the files. Remember to change the location we use in this chapter to your location.

Note: Update these files in the repository to reflect your location:

- `server.sh` (the content of this file points to the location of `wsclient.html`, and should point to a location on the computer)
- `server.sh = cat /Users/drice/Documents/websocket/index.html`
- `index.html` (which is included in the repo)

Importing a NiFi Template

When you have NiFi successfully installed and running, you can import the `HelloWorld.xml` file, which is a NiFi template cloned from the GitHub repository (see the sidebar on how to clone the GitHub repository).

Follow these steps to import a NiFi template (see Figure 2-1):

1. **Launch NiFi by pointing your browser to this location `https://localhost:8443/nifi/` (or a similar location based on how NiFi was installed).**
2. **Click the Upload Template button within the Operate window.**
3. **Click the magnifying glass icon.**
4. **Browse to the location where you downloaded the GitHub repository, and select the `HelloWorld.xml` file.**
5. **Click Open and then Upload.**

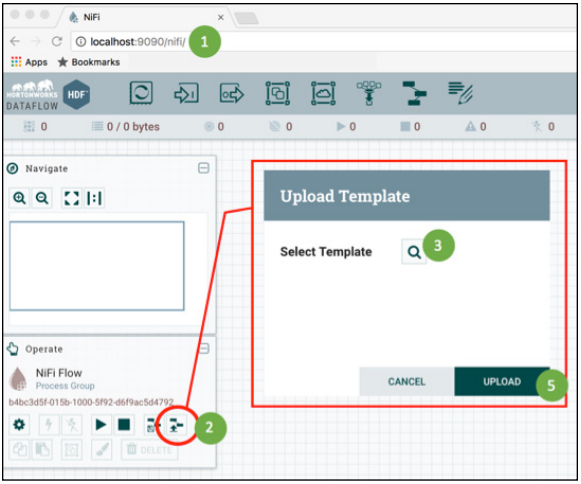

FIGURE 2-1: Importing the `HelloWorld.xml` NiFi template.

TIP

Templates are considered deprecated in the last several releases of NiFi and are scheduled for removal in NiFi 2.0. Flow Definitions, represented in JSON, are the recommended way of importing and exporting reusable flow configurations.

You can export a process group as a Flow Definition starting from NiFi 1.11 by right clicking and selecting Download Flow Definition. To import a Flow Definition, drag the process group icon to the canvas and click the Upload icon next to the text field.

Adding a NiFi Template to the NiFi Canvas

With the NiFi template uploaded, you can add the Hello World template to the NiFi canvas by following these steps (see Figure 2-2):

1. **Click the Template icon in the grey navigation bar at the top of the screen and drag and drop it anywhere on the canvas.**
2. **In the Add Template window, choose the Hello World template.**
3. **Click the Add button.**

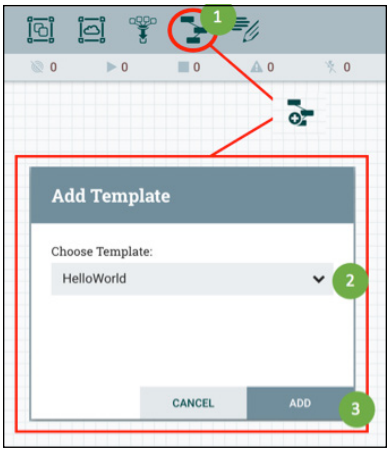

FIGURE 2-2: Adding the `Hello World` template to the NiFi canvas.

The Hello World dataflow opens on your canvas, as shown in Figure 2-3. The NiFi processors have a yellow exclamation icon next to them. You need to modify these processors (along with the `PutFile` processor and the `ExecuteStreamCommand` processor) for the Hello World dataflow to work.

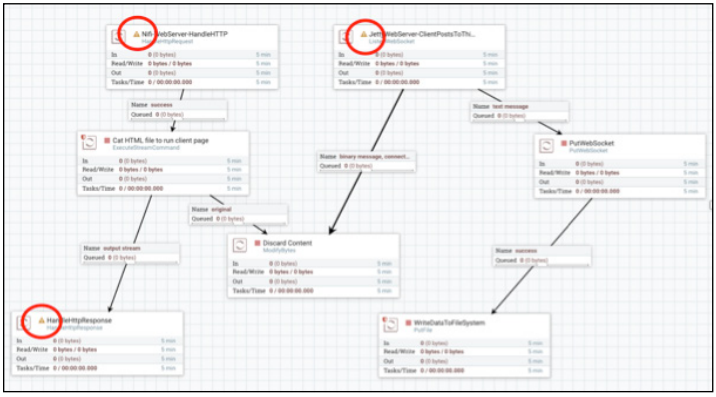

FIGURE 2-3: The Hello World example dataflow.

Setting Up and Running the Hello World Example

With the Hello World template added onto the NiFi canvas, you can clean up some properties and configurations within the processors that have a yellow exclamation icon next to them. The yellow exclamation icon indicates those processors need attention.

Configuring HandleHttpRequest NiFi processor

Start by correcting the `HandleHttpRequest` NiFi processor by following these steps:

1. **Right-click the `Nifi-WebServer-HandleHTTP` NiFi processor and select Configure.**
2. **In the Configure Processor window, click the Properties tab.**
3. **Click the right-pointing arrow in the third column next to `StandardHttpContextMap` (see Figure 2-4).**

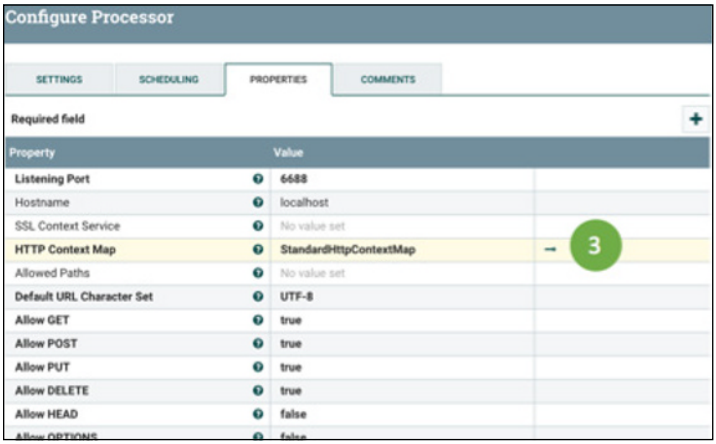

FIGURE 2-4: Configure the `HandleHttpRequest` processor.

4. **In the Process Group Configuration window, click the lightning bolt on the same row as the `StandardHttpContextMap` to enable this controller service (see Figure 2-5).**

FIGURE 2-5: Enable the `StandardHttpContextMap` controller.

5. **Leave the Scope as Service Only and click Enable and then Close.**
6. **Close the Process Group Configuration window.**

With the `HandleHttpRequest` NiFi Processor configured properly, there is now a red box next to it, indicating the processor isn't running.

Configuring the other processors

When you know how to configure the `HandleHttpRequest` NiFi processor, you can easily configure the other processors needed to run the Hello World dataflow: `ListenWebSocket`, `PutFile`, and `ExecuteStreamCommand`. Just as you do with `HandleHttpRequest`, right-click the processor you want to configure, and select Configure. Then on the Properties tab of the Process Group Configuration window, change these settings for each processor:

- **`ListenWebSocket`**: Click the right-pointing arrow in the third column next to `JettyWebSocketServer`. Click its lightning bolt to enable the service. Select Service Only for the Scope. Then click the Enable and Close buttons.
- **`PutFile`:** Change the Directory property to the directory where you want the data from the web application to write to and click the Apply button.
- **`ExecuteStreamCommand`:** Change the Command Path property to the directory where you downloaded the NiFi HelloWorld GitHub repository and extracted the `server.sh` file. Then click the Apply button.

When you have the processors configured correctly, each has a red box next to it, indicating the processor isn't running, instead of the yellow exclamation icon.

Running the Hello World example

To start the Hello World flow, follow these steps (see Figure 2-6):

1. **Click anywhere on the NiFi canvas so that nothing is selected (meaning no processor or connection is highlighted).**
2. **Click the Start button on the left side of the Operate window.**

   Every red box for each processor changes to a green triangle pointing to the right, indicating the processor is running.

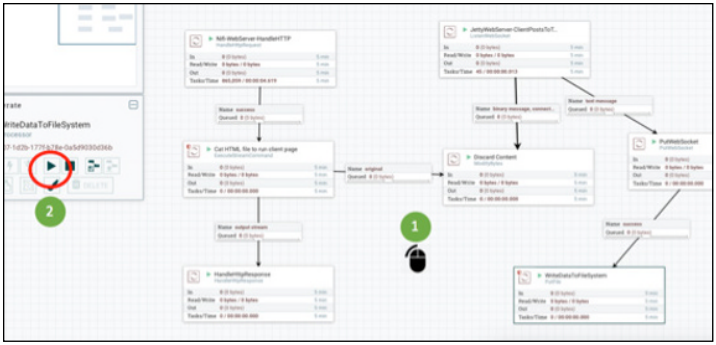

FIGURE 2-6: Running the Hello World dataflow.

Now you can launch a web browser that NiFi is hosting. Follow these steps (see Figure 2-7):

1. **Point a web browser to** `http://localhost:6688/`**.**

   You see a text box along with three buttons: Open, Send, and Close.
2. **Click the Open button.**
3. **Enter** Hello World **in the text box.**
4. **Click the Send button.**

   Congratulations, you just submitted Hello World through your NiFi flow.

To validate that Hello World flowed through NiFi, open the destination path defined in the `PutFile` NiFi processor and you see a file that contains the phrase "Hello World" inside it.

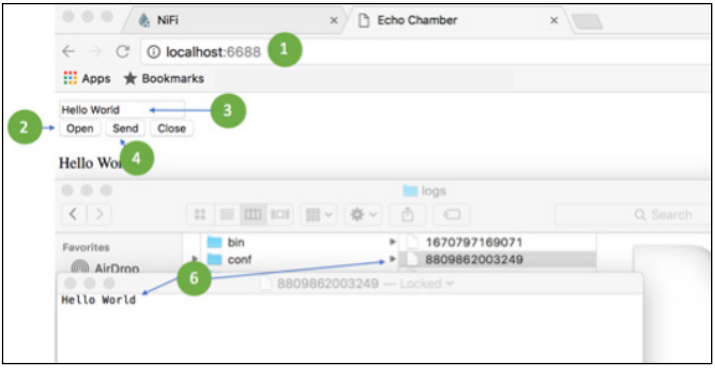

FIGURE 2-7: Executing the Hello World demo.

Understanding the Hello World Example

Now that you've successfully executed the Hello World dataflow, you can better understand how it works behind the scenes.

The `HandleHttpReqest` NiFi processor starts an 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 processor, which executed the `server.sh shell` script.

This shell script launches the HTML file inside of NiFi. Like any web server, the call is then routed to see whether the response is successful and this is how the HTML file is presented in the web browser.

The content of the `index.html` file is in a simple form. The Open command established a connection from the web client to the `ListenWebSocket` connection over port `9998` (the `ListenWebSocket` was pre-configured to listen to this port when you imported the NiFi template). After the connection is established, you can publish any data over the web socket connection.

Eventually, the data put on the `WebSocket` connection is routed to the `PutFile` processor within NiFi where the data is stored on a local disk in the directory you specified.

IN THIS CHAPTER

- Getting information through the NiFi's user interface
- Setting up backpressure to help processes run smoothly
- Debugging with provenance
- Getting information from the NiFi server logs

Chapter 3
General Debugging & Monitoring

The last thing anyone wants is for a processor to stop running unexpectedly, which prevents it from completing or preventing others from running altogether. Fortunately, NiFi offers several methods to monitor them.

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 debugging efforts when things don't go as planned, and monitor 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 process 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.


TIP

Funnels are notoriously used as anti-patterns to store flowfiles that may not be expiring. While valid in development, such setups can back up and cause other operational issues. 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, as shown in Figure 3-1.

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

- The **JVM** tab shows metrics about on and off-heap utilization and garbage collection counts along with total time.

- **System,** as shown in Figure 3-2, 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.

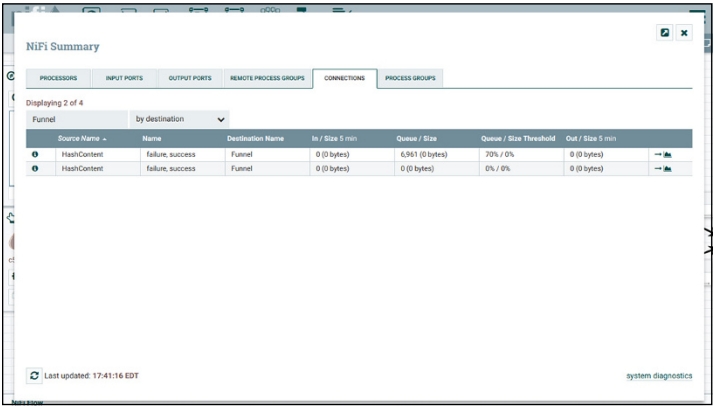

FIGURE 3-1: You can keep track of funnels through the Connections tab.

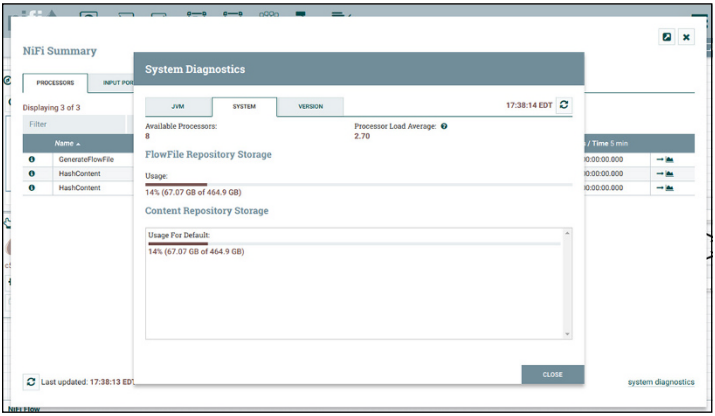

FIGURE 3-2: The system storage used for the NiFi repositories.

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 in Figure 3-3, can visualize many metrics related to the processor or connection and includes separate plotting for each NiFi node in the cluster.

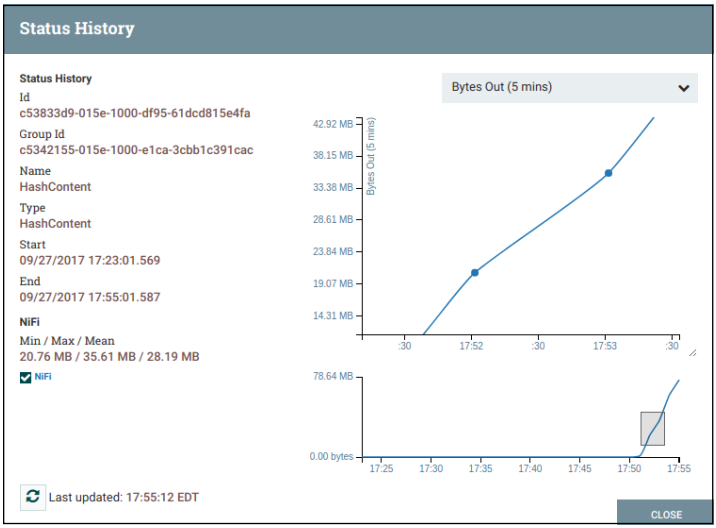

FIGURE 3-3: The graph and information that you can find via the Status History menu.

TIP

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

Backpressure

From an operational perspective, backpressure enables you to design a system that self-regulates the amount of storage it's utilizing to prevent it from crashing! Backpressure isn't typically thought of as a first-class citizen in many data movement systems, but NiFi provides it as a first-class feature.

Understanding how backpressure works

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 backpressure 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.


TIP

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.

Configuring backpressure

In NiFi, backpressure is configured at the connection level, where you can manage the backpressure policies. You can configure backpressure 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. When filled, the upstream processor pauses processing, allowing the system to catch up.
- **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.

WARNING

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 the normal operation of the NiFi server that is 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

TIP

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 application. 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 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 attributes, as well as the queue positions and durations, along with the node that performed the event. The Content tab allows you to investigate before and after versions of a flowfile after it's been processed (see Figure 3-4); 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.)

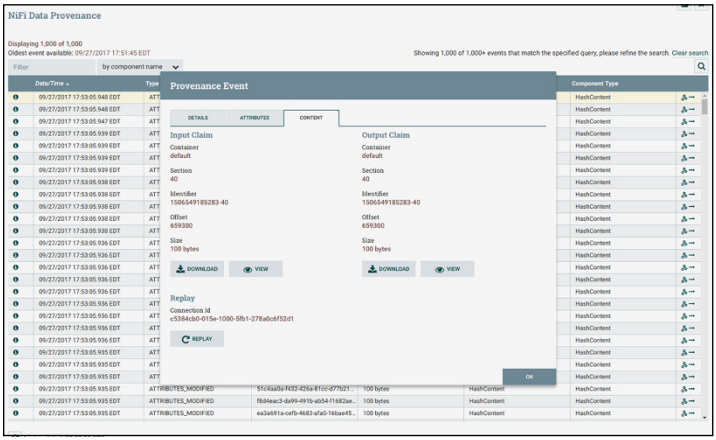

FIGURE 3-4: A major advantage to provenance is the capability to view the before and after the content of a flowfile.

Checking the NiFi Server Logs

Each NiFi server has a set of application and bootstrapping logs. NiFi uses Logback 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 log level for NiFi is set to `INFO` by default. Users can customize the log level (`WARN`, `DEBUG`, `ERROR`) for better debugging.


TIP

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 cluster-wide configuration. For example, to only change the processor log level, edit the `logback.xml` and change the logger line for `org.apache.nifi.processors` from `WARN` to `INFO`.


TIP

Alternatively, the log level can also be changed at the processor level using the UI. However, this may help only during development or debugging using the UI or when you don't have access to the node-level logs. Remember to set it back to the default when you're done.

IN THIS CHAPTER

- Importing data into a database
- Pushing the data over an HTTP connection
- Polling and saving data for later use

Chapter 4
NiFi Use Cases

Planning the first NiFi data integration project requires attention to:

- **Data volume and velocity:** Pulling flat files from a monitored directory 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 binary and text data, either in a structured or unstructured format.
- **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 available, 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 another destination, (such as Apache Kafka), local disk storage, or cloud object storage (such as Amazon S3).

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 4-1.

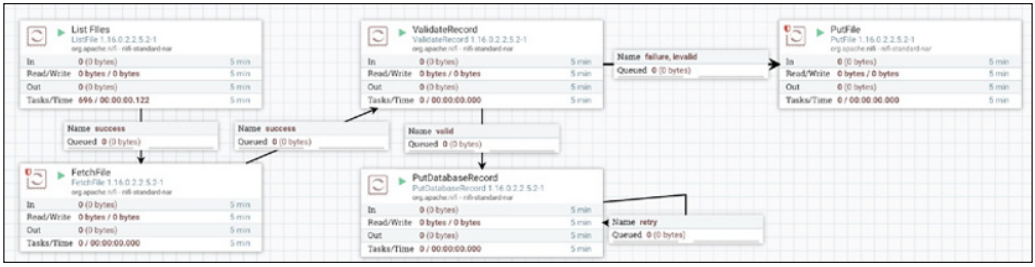

FIGURE 4-1: The steps to this dataflow.

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 to include the mapping of field names in the data to database column names.


TIP

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

Listening for HTTP Posts

In this scenario, an external system pushes the data over an HTTP connection into NiFi using a `POST` option. The approach to setting up a listener operation is as simple as inserting the `ListenHTTP` processor, as shown in Figure 4-2.

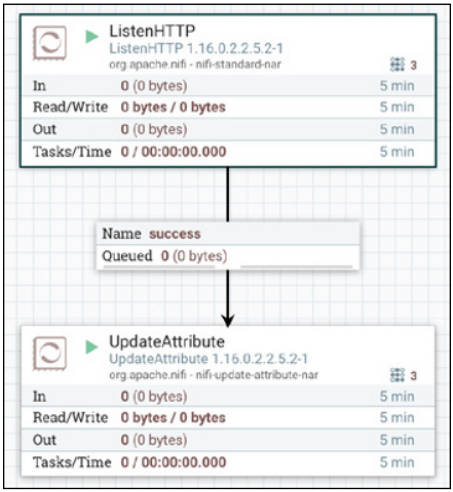

FIGURE 4-2: This flow inserts a `ListenHTTP` processor.

This flow is listening on the same server the NiFi process is running on and listening to a port configured in the `ListenHTTP` processor, as shown in Figure 4-3.

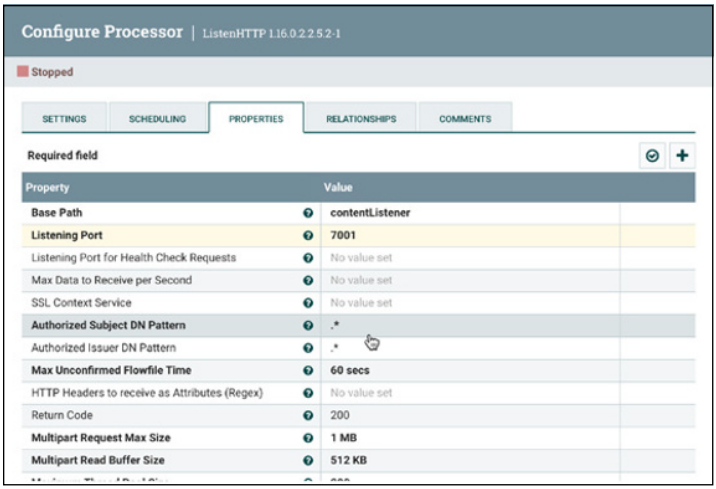

FIGURE 4-3: The `ListenHTTP` processor includes a port for listening.

With this configuration, any HTTP `POST` operation to the NiFi server using the URL `http://{nifi server address}:7001/contentListener` gets picked up by the listener and then gets passed on to the next step in the dataflow.

REMEMBER

The capability to specify a base path (the example in Figure 4-3 specifies `contentListener`) means that it's possible to have multiple readable HTTP endpoints for all the different workflows supported by your NiFi instance.

Polling a RESTful API to Extract a JSON Attribute

In this scenario, the flow executes `RESTful` queries every 30 seconds and then saves the result as a NiFi attribute for use in other downstream functions, as shown in Figure 4-4.

The first step in this scenario is to poll the `RESTful` API to query the Yahoo Weather service every 30 seconds and return a flowfile in JSON format. To control the polling interval, specify 30 seconds for the `Run Schedule` property on the Scheduling tab.

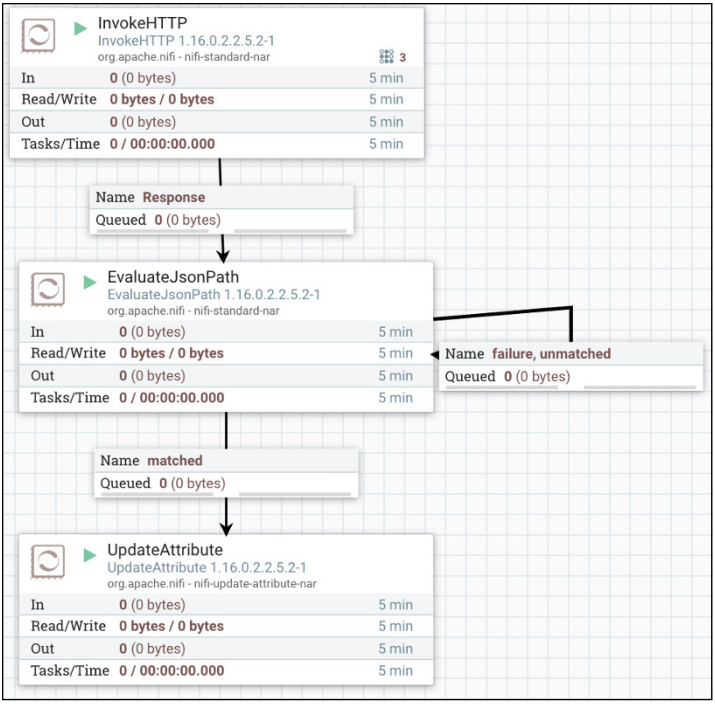

FIGURE 4-4: This flow shows a `RESTful` query runs every 30 seconds and saves the result.

Without any programming, you can specify an HTTP URL on the Properties tab to make the Yahoo Weather query.

To make this use case work, you need to extract the wind speed from the full JSON object. The data returned by the `InvokeHTTP` processor is actually a fairly complex JSON object. This is a problem because the objective is to only pull out the wind speed attribute from within this large JSON object.

To extract just the `wind speed` JSON attribute, you need to add a NiFi attribute using the `EvaluateJsonPath` processor, as shown in Figure 4-5.

To add it, click the + sign in the top-right corner of the `Evaluate JsonPath` processor Properties tab and add the `wind_speed` attribute. Looking at the output from the `EvaluateJsonPath`, you find a new attribute with an actual `flowfile-content` value.

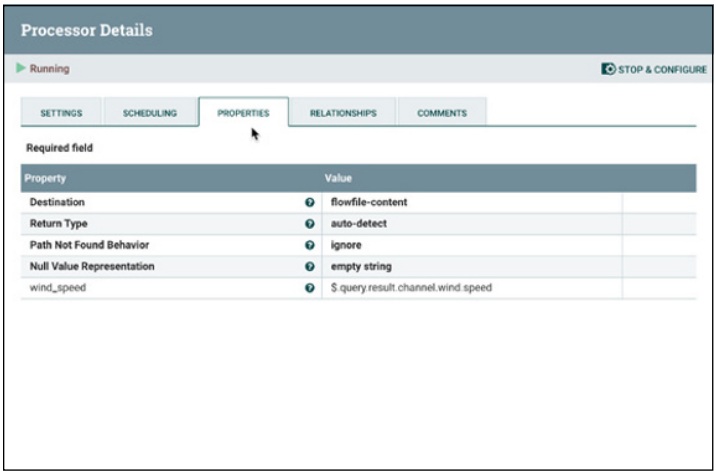

FIGURE 4-5: Use `EvaluateJsonPath` processor to extract the wind speed.

IN THIS CHAPTER

- Flow overview
- Setting up flow layout
- Properly load balancing
- Scheduling and threading flows
- Optimizing flows

Chapter 5
NiFi Anti-Patterns

Anti-patterns are routines that are frequently found in dataflows, but they end up not being effective, and may even not be productive at all.

In this chapter, we go over common NiFi anti-patterns found in dataflows. Knowing these anti-patterns can help you improve the design and layout of your flow, optimize processors and connections with scheduling and load balancing, and optimize your dataflow for best performance.

TIP

If you want to learn more about Apache NiFi anti-patterns, see Chapter 8 for an additional resource.

Flow Overview

There are three Nifi anti-patterns that occur in NiFi flows:

- **Splitting a flowfile into parts and then merging them together again.** This split and merge also affects the lineage and makes it very hard to look at data provenance.

  The solution is to use NiFi's record-based processors (`SplitText` processor and `AttributesToJson` processor are two) with parsers versus splitting new lines.

- **Using Regex matching to unstructured text.** Avoid regular expressions where possible. They are hard to write and often do not actually solve complicated structured data.
- **Blurring the line between flowfile attributes and content.** You need to keep flowfile content and flowfile attributes separate.

The solution for these three anti-patterns is to use NiFi's record-based processors with parsers versus splitting new lines, using regular expression to match unstructured data, and moving data from content to attributes and back again.

Record-based processors include record readers and record writers. Two common ones are `QueryRecord` and `PutRecord`. These readers and writers can be a different type and schema than the source data. Additionally, when using record-based processors, the data provenance and lineage are now navigable within a data provenance UI.

TIP

A query in these processors can include additional columns calculated by NiFi functions or flowfile attributes.

Laying Out Flows

Properly laying out your flow is very important to be able to visually follow the flow of data. Creating a proper flow that is easy to follow greatly helps when trying to operate or debug the flow. Any developer who inherits your flow needs to be able to visually understand where data is coming from upstream, what is happening to that data in the branches of your dataflow, and where it's going downstream.

WARNING

Avoid spaghetti flows. You don't want to end up in a situation depicted in Figure 5-1.

REMEMBER

Every dataflow is unique, but taking a consistent approach to flow layout and using all the NiFi UI tools make your dataflow easier to work on for yourself and others who may use it later in its lifecycle.

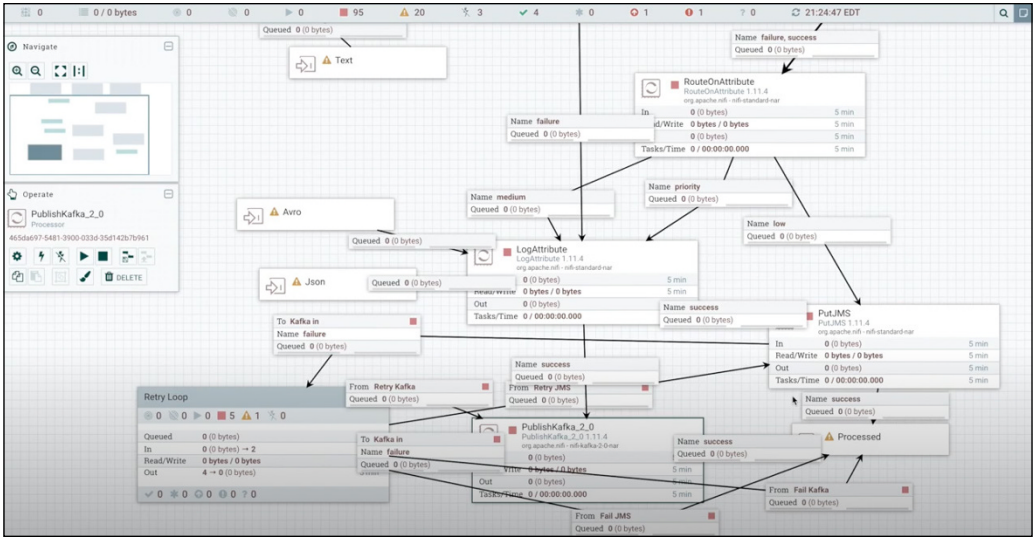

FIGURE 5-1: Avoid tangled and confusing dataflows.

Arranging flow direction

Top to bottom or left to right are the most common dataflow structures. Whichever path you take, you should be consistent, and organize the flow accordingly. Create a dataflow path that is visually traceable.

Here are some tips to achieve better dataflow structure:

- Use top to bottom where possible, and align these as close as possible within the NiFi canvas.
- Create bend points for connection lines that are overlapping other UI elements.
- Combine connections where appropriate, by creating one connection with multiple relationships.


TIP

You can use process groups with input/output ports to create advanced flows.

Naming and commenting processors

Naming and commenting on processors is very useful. Rename your processors to something appropriate for their function. For example, ReplaceText could be Fix Bad Characters.

Every processor includes a Comments tab. It's a great place to provide comments to yourself or others that give a greater explanation of what the processor does.


TIP

Click the black arrow at the bottom right of any processor to show its description.

Using processor labels

At the top right of the NiFi UIis a Labels icon. Dragging a label onto the screen allows you to create descriptive text boxes on your NiFi canvas. These labels are great for providing directions, information, or documentation important to your flow.


TIP

You can change the color and text size of your labels to suit your needs by right clicking on the label and selecting Configure to rename or clicking on the bottom right corner and dragging to resize.

Balancing Loads Correctly

Load balancing is the process of taking data on one NiFi node and spreading it across the entire cluster — thus parallelizing the processing of that data.

WARNING

NiFi does not automatically distribute data between nodes.

You need to intelligently decide when to distribute data across the cluster and when not to. How do you know when? The answer is found by evaluating if it is economical to move data across the entire cluster to other NiFi nodes or leave it where it already is on a given node.

Properly load balancing your flow makes your flow operational from 100 to 1,000 plus nodes at rates up to billions of transactions per second. To achieve this type of NiFi performance, you want to avoid these three anti-patterns:

TIP

- **Load balance compression:** This option should be set to Do Not Compress At All.

  Look at the bottom right of the queue for the configuration summary.

- **Overload balancing within your flow:** When you distribute the load at the top of a flow, you do not need to configure deeper queues for load balancing; the data is already distributed upstream. These deeper connections should remain Do Not Load Balance.

WARNING

  Make sure your top-of-flow processor isn't already distributing its load around the cluster. For example, Kafka processors already have distribution built in.

- **Incorrectly using the Primary Node feature of a processor after load balancing:** Instead use the primary node at the top of your flow and then load balance on the next processor. For example, use `ListFile` (set to primary node), configure Load Balance on the next connection to `FetchFile`, and then process further down the flow as normal.

Scheduling

WARNING

Getting scheduling wrong affects the performance and the stability of your cluster.

Your flows are operational from 1,000, 100,000 or 1,000,00 transactions per second on each node in your cluster.

You should find a sweet spot during flow tuning. After you're sure the flow is operating correctly, you can then increment your thread pool values, re-evaluate your flow, and retune in a new iteration.

TIP

Do not overtune any of these values by jumping from small to very large values. If everything is tuned correctly, and a processor is struggling to keep up with performance demand, take a look at the disk configuration.

Thread pools

Thread pools affect all users and dataflows. Configure them in the controller settings that are found in the top-right menu. There are two different thread pools:

- **Event driven:** This is where you tell NiFi how many different processors you want to run at any given time.

  Set this between two and four times the number of CPU cores available.

  This setting is per node, so don't set the total number of cores by two to four times the number of nodes.
- **Timer driven:** Leave this set to 1.

WARNING

Processor scheduling

Configure processor scheduling within your processors and dataflow connections before increasing thread pools. From the Scheduling tab of a processor, you can change these settings:

- **Concurrent Tasks:** This is the maximum number of threads the processors are able to use.

WARNING

Use this setting sparingly. Increasing settings to large numbers without evaluating other tuning methods is also an anti-pattern. Increase in small iterations.

- **Run Duration:** This slider bar configures micro batching. Instead of increasing concurrency, increase the run duration.

CPU utilization

To view CPU utilization, go to the top right menu and choose Summary➪ System Diagnostics➪ System Tab. Look at total cores and the load average. If there is some *overhead* (the load isn't equal to cores), increase the size of the thread pool slowly. Use incremental iterations where you evaluate the flow on each iteration.

WARNING

Be sure to leave some overhead to account for nodes dropping out of the cluster. In some cases 50 percent is right; in others, 70 percent to 80 percent is better.

Optimizing Flows

In this section of common NiFi anti-patterns, we cover how to properly use the Primary Node Only feature, when to use the `ConvertRecord` processor, how to avoid complex expression language usage, and why to avoid using `LogAttribute` in your dataflows.

Primary Node Only

Using the Primary Node Only feature of any processor should be carefully considered. In most cases a Primary Node Only processor should be at the start or top of your dataflow. Setting deeper dataflow processors to the primary node can create a situation where data is on the primary node when the primary node changes. When this happens the data can not be moved to the new primary node.

REMEMBER

Only source processors (top of dataflow) should be set to Primary Node Only.

ConvertRecord

Overusing `ConvertRecord` is another common NiFi anti-pattern. There are three scenarios often seen in NiFI dataflows that use `ConvertRecord`:

- **When to use a `ConvertRecord`:** Best practice is to use `ConvertRecord` when there are no other `ConvertRecord` processors in a flow.
- **When not to use `ConvertRecord`:** Do not use a `ConvertRecord` processor when you're already using a record-based processor. Remove the `ConvertRecord`, and do the appropriate conversion in the source processor.
- **Avoid the overuse of `ConvertRecord`:** Don't use a record-based processor, then a `ConvertRecord`, and then yet another record-based processor. Overusing causes performance issues.

Using complex expression language

The next NiFi anti-pattern to cover is using complex expression language in NiFi processor configuration. Complicated `if-then-else` expression language chains are very hard to write, very hard to understand, and can often be replaced with NiFi expression language functions.

Follow these tips when using expression language:

- Press Shift+Enter to create new lines to make chained expressions easier to follow.
- When possible, evaluate other NiFi functions that can provide the same end result.
- Press Control+Space to show expression language functions with hover descriptions in the UI.

TIP

Right-click any processor to link directly to the documentation for that processor.

NiFi EXPRESSION LANGUAGE USER GUIDES

There are so many NiFi expression language functions that NiFi dataflow developers should be aware of where to find the documentation. You can find this documentation on the Apache NiFi website, as well as in your NiFi user interface in the Help menu. Be sure to check out the Expression Language Guide and Admin Docs that ships with every instance of NiFi. One advantage of having this information local to your instance of NiFi is the docs are specific to your version of NiFi.

Logging attributes

Logging attributes as a debug method is an anti-pattern that can greatly impact the performance of your flow. This behavior puts a large number of log output lines into the NiFi log file preventing the ability to look at those logs for other purposes. The preferred alternative to debugging within a dataflow is to use NiFi Data Provenance in the NiFi Admin menu. It allows you to view the entire lineage of a flowfile, including all its attributes and content at any point in the dataflow. This method provides a much greater perspective when compared to trying to use log attributes and hunting through the logs. You can also search for filenames and attributes.

REMEMBER

During a debug event, if changes are made to the dataflow, you can use the NiFi Data Provenance UI to replay those flowfiles.

IN THIS CHAPTER

- Leveraging the benefits of record-based processors
- Finding out the different types of processors

Chapter 6
Record-Based Processors

Traditionally, NiFi didn't care about the content of data. A flowfile is just "data" whether it's an image, text without structure, or text in JSON. This powerful characteristic makes NiFi efficient, data agnostic, and suitable for handling unstructured data such as Internet of Things (IoT) data. However, a lot of enterprises use NiFi to manipulate structured data in different formats: CSV, JSON, and Avro to name a few. In this chapter, we show you how to use record-based processors with structured data.

The Benefits of Record-Based Processors

To make managing structured data easier, record-based processors were first introduced in NiFi 1.2. Since then they're constantly improved and new processors added. Before record-based processors, developers had to deal with the following issues:

- Files were split and operations were done at the individual record/event level, which could cause performance issues when files sizes were large.
- NiFi flowfile attributes would be replicated across each file split, causing additional memory overhead. Keeping the

file and underlying record as a whole can provide performance and operational benefits to NiFi flow processing.

- Files were split and operations were done at the individual record/event level. This can cause some performance issues as some files can contain millions of records. The metadata is replicated across all records, creating additional memory overhead.

Being able to keep the record as a whole can give some serious performance advantages:

- Record-based processors offer better performance, functionalities, and ease of use.
- They can help reduce the overall flow complexity (Split, Evaluate Json Path, Update Attribute, for example) and help simplify your canvas.
- They also lead to consolidating the reusable components, thereby increasing the overall NiFi cluster efficiency.

Record-Based Controller Services

Record-based processors leverage a set of *deserializers* (record readers) and *serializers* (record writers) to efficiently read, transform, and write data. They leverage c*ontroller services*, which are reusable components contained within the scope of the NiFi canvas. The record readers and writers infer the schema, get the schema from NiFi's internal Avro Schema Registry, or use an external schema registry such as the Cloudera Schema Registry.

REMEMBER

Auto inference of schema simplifies schema management but could cause data mismatch with target systems such as RDBMS or SQL engines such as Hive. This could potentially cause a mismatch with a table DDL's causing `NULL` values in your table.

Use of schema registry allows you to have control over schema and reject records not conforming to schema rules. It also allows easier evolution/transition of the schema as the upstream/downstream data changes.

NiFi supports many types of record-based processors but they can be mainly classified into three types: Source, Sync (Target), and Transform.

Source

Source record processors can contain only a record writer (RDBMS processors such as `Query Database Table Record` and `Execute SQL Record`) or contain both a record reader and record writer (such as `Consume Kafka Record`) when the source of the data vary. The difference between the RDBMS processors and others is that data output is in Avro format, thus normalizing the data type. Other record processors manage data in Kafka topics, support multiple formats (for example, CSV and JSON), and are ingested based on the source topic datatype.

Sync (Target)

Sync (Target) processors consist only of a record reader for RDBMS processing. For processing of other sources such as `Publish Kafka Record Processors`, target processing is similar for both record readers and record writers.

Transform

Transform record processors come in a few different varieties:

- Data type transforms such as the `ConvertRecord` processor that converts one data type into another.
- The `JoltTransformRecord` processor that can manipulate data, including flattening.
- The `QueryRecord` processor that does SQL operations against the flowfile.
- The `UpdateRecord` processor that allows you to make changes to various keys/values across the entire record.

WARNING

A common mistake seen with typical NiFi flows are the constant splitting of flowfiles, placing data in attributes, modifying with the expression language and then creating content from those adjusted attributes. A better approach would be to use the `Update Record` processor that allows you to modify flowfile content much more efficiently without splitting the flowfiles.

IN THIS CHAPTER

- NiFi Registry
- Stateless NiFi
- KConnect
- Cloudera DataFlow
- Cloudera DataFlow Functions

Chapter 7
Other NiFi Features

NiFi has many other features and functions than what has been mentioned in the prior chapters. So in this chapter, we decribe some of the more interesting features you might find useful when using Apache Nifi.

NiFi Registry

NiFi Registry, a subproject of Apache NiFi, is a service that enables sharing of NiFi resources across diverse environments. Registry is a uniquely coupled complementary application to Apache NiFi, providing a central location for storing, managing, and enabling version control of data flow resources.

When designing a dataflow in NiFi, you may need to develop and test different approaches and build incrementally towards finalizing your flows. This is where Registry can help you with the full Flow Development Lifecycle (FDLC), shown in Figure 7-1. It seamlessly integrates with NiFi, facilitating storing, retrieving, and upgrading versioned flows. NiFi Registry helps you track multiple flow versions and easily navigate among them.

REMEMBER

NiFi Registry can be an essential component to controlling and managing your data pipeline and flows in a DevOps environment to accelerate the timeline from flow development to operationalizing your flows.

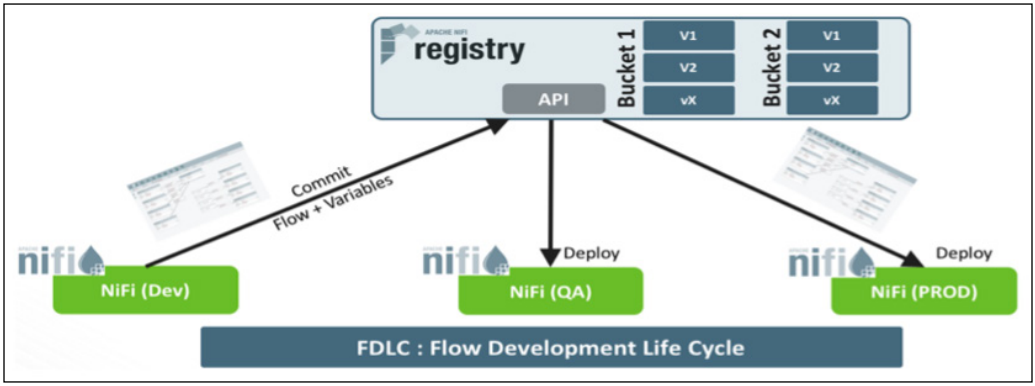

FIGURE 7-1: Managing flows with NiFi Registry.

Versioned flows are stored and organized in containerized *buckets* (nested containers for storing and organizing versioned flows) within the registry. Buckets can be aligned to various teams, specific to deployment environments, based on usage patterns, or be assigned to different projects or business/mission units.

Key features of NiFi Registry

With NiFi Registry, you can

- Access it from a web UI (default port is 18080).
- Use the REST API, which provides an interface for flow registry operations.
- Initiate version control of flows directly from the Apache NiFi canvas.
- Export/import versioned flows across NiFi deployments (for example, Development ➪ QA/Staging ➪ Production).
- Commit and/or revert local flow changes.
- Support user authentication methods including LDAP, Active Directory, and Kerberos.
- Manage users and groups level permissions for access control to flows and buckets with role-based-access (RBAC) policies.

Getting started with NiFi Registry

Details for downloading and installing NiFi Registry are beyond the scope of this book. However, you can find additional information in the Apache NiFi Registry documentation.

After you've downloaded and installed NiFi Registry,you can start the same way you'd start any application.

To set up NiFi Registry, follow these steps:

1. **Open a web browser and navigate to `http://localhost:18080/nifi-registry`.**

   The default web UI port, shown in Figure 7-2, is 18080 and, if needed, you can change it by editing the `nifi-registry.properties` file in the NiFi Registry conf directory.

FIGURE 7-2: The web UI for NiFi Registry.

2. **Create a new bucket to store versioned flows.**

   a. *Click the wrench icon in the upper-right corner of the UI.* A pop-up window listing the currently configured buckets appears.

   b. *Click the New Bucket button, name your bucket, add a description (optional), and click Create.* See Figure 7-3.

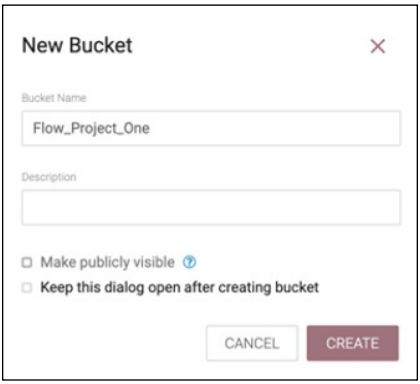

FIGURE 7-3: Create a new bucket in NiFi Registry.

3. **Connect a NiFi instance to Registry.**

   a. *Choose Controller Settings (see Figure 7-4) from a currently running NiFi instance and add Registry Client under the NiFi Setting tab.*

   b. *To add a Registry Client, click the plus sign icon. In the pop-up window, enter the Registry information and then click Add.* See Figure 7-5.

4. **Initiate version control management at the process group level in a running NiFi instance.**

   a. *Right-click a process group and choose Version ➪ Start Version Control from the context menu, as shown in Figure 7-6.*

b. *Fill in the details in the Save Flow Version box, shown in Figure 7-7, and click Save.*

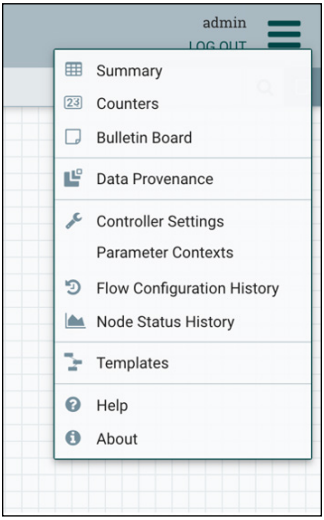

FIGURE 7-4: The global menu with Controller Settings.

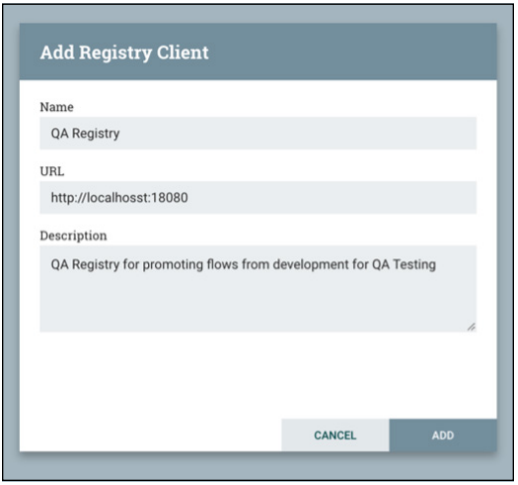

FIGURE 7-5: Add a Registry Client.

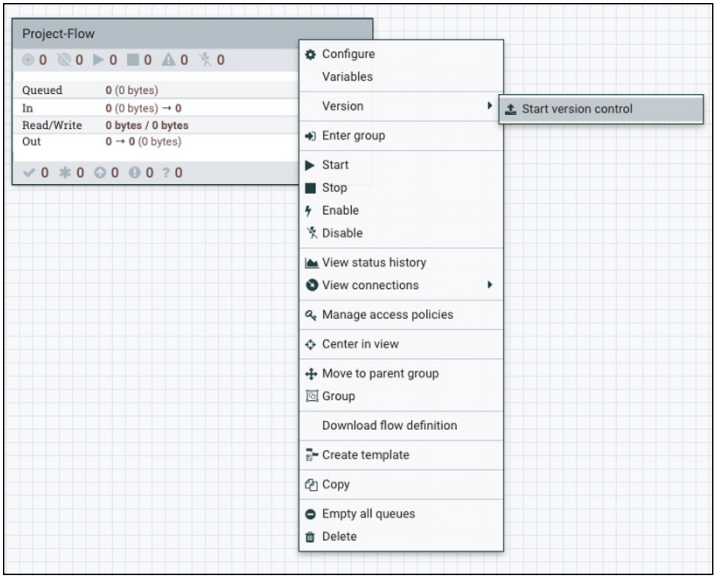

FIGURE 7-6: The Start Version Control option.

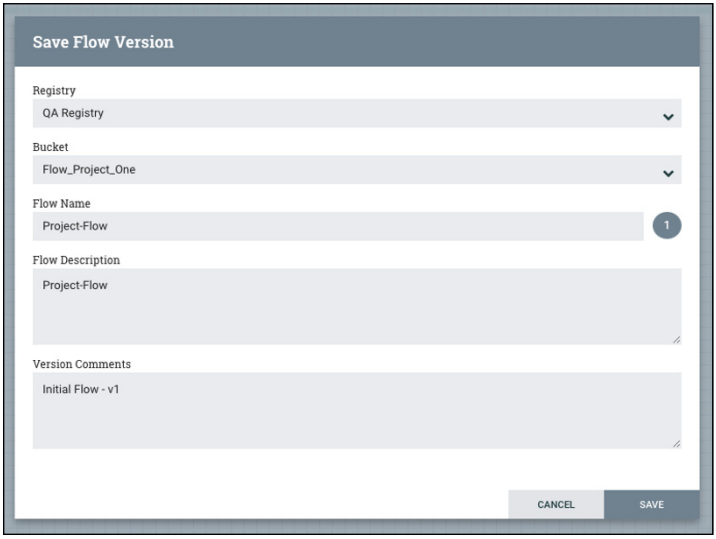

FIGURE 7-7: Save the flow version for a progress group.

Once the version control attributes are saved, the process group is now under version control as denoted by the green checkmark in the upper left corner of the Process Group, as shown in Figure 7-8.

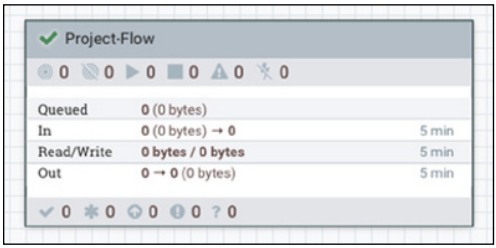

FIGURE 7-8: This progress group is under version control.

When working with NiFi Registry, keep these tasks in mind:

- If there are any uncommitted changes to the process group or its underlying flow processors, then the version control icon changes to an asterisk symbol, as shown in Figure 7-9.
- To manage the version control actions for the process group, right-click the process group and select the appropriate action from the context menu, as shown in Figure 7-10.
- To view the changes to the process group, navigate to the Registry UI and from the main page click the versioned flow bucket name (a refresh may be required). You can view the log for the project flow, as shown in Figure 7-11.

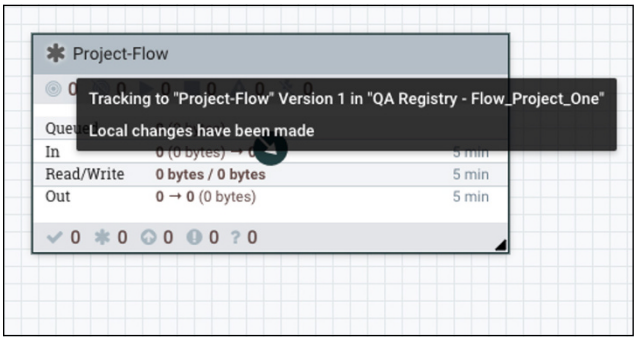

FIGURE 7-9: This process group has uncommitted changes.

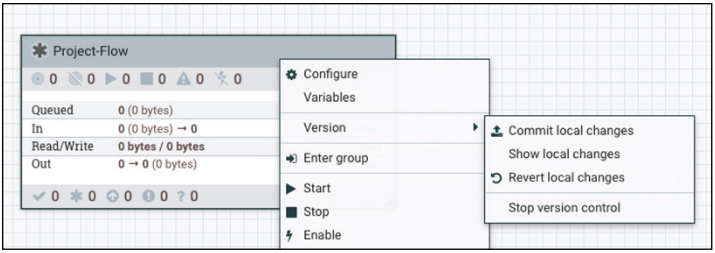

FIGURE 7-10: The Version Control Action menu.

FIGURE 7-11: The change log for the project flow.

You can find more information about NiFi Registry in these guides:

- NiFi Registry User Guide (`https://nifi.apache.org/docs/nifi-registry-docs/html/user-guide.html`): A basic user guide for accessing the Apache NiFi Registry. Topics include managing flows and buckets, administering users/groups, along with configuring special access and privileges for users and groups.
- NiFi User Guide (`https://nifi.apache.org/docs/nifi-docs/html/user-guide.html`): This guide provides information on how to navigate the Registry UI and explains in detail how to manage flows/policies/special privileges and configure users/groups when the Registry is secured. Topics include connecting to a registry, version states, importing a versioned flow, and managing local changes.

Stateless NiFi

NiFi was originally designed to run as a large, multi-threaded, multi-tenant application. Data is checkpointed and saved to disk across the different flows to reliably deliver data to the destination.

However, NiFi is used in a particular use case to route data from a Kafka consumer to a Kafka producer. In this case, NiFi needs to support exactly once semantics. The default NiFi engine has a few obstacles that prevents this case.

- Because NiFi persists all data across restarts, when restarted, it can send the processed data again, which can cause data duplication.
- NiFi is loosely coupled. Exactly once semantics require that the producer and consumer be tightly coupled so that when there is a problem, entire data can be rolled back and processed again.
- For complex use cases where data in NiFi is split and re-ordered, it becomes difficult to support exactly once semantics.

NiFi Stateless architecture is designed to prevent these obstacles and support exactly once semantics. It's single-threaded, and it can optionally keep the data in memory to make processing fast but no data is persisted across restarts. It also runs the entire dataflow as a single cohesive unit; if there is an issue, the entire transaction is rolled back. The data is never re-ordered between transactions.

ExecuteStateless processor

You can use the `ExecuteStateless` processor, new in NiFi 1.15, to run a stateless engine from traditional NiFi.

To use the `ExecuteStateless` processor, you need to first create a process group containing the flow you want to execute with the stateless engine. You would then download the flow definition, or commit the flow to a NiFi Registry instance. From there, you would configure `ExecuteStateless` with the location of the flow definition. When configuring the process group, the `Parameter Context` property is set to `Enrich Changes` and the `Outbound Policy` property is set to `Batch Output`.

You need to make configuration changes for the Kafka consumer and publisher so that data is sent as a single transaction:

- `ConsumeKafkaRecord`: Set `CommitOffsets` to `False`, so that if there is a processing failure the entire transaction can be rolled back.

- PublishKafkaRecord: Set Use transactions to True. When the consumer doesn't commit offsets, it updates the flowfile with an attribute so that the publisher knows that it's responsible to commit offsets. PublishKafkaRecords needs to be out of the process and needs to be connected from the success connection of the process group.

Stateless KConnect Source and Sinks

For deployments where NiFi acts only as a bridge to transfer data into and out of Kafka, it may be simpler to have the dataflow run within Kafka Connect.

The NiFi Kafka Connector allows you to do just that!

The dataflow runs with Kafka Connect using the stateless NiFi dataflow engine. For more information, see the earlier "Stateless NiFi" section.

NiFi supports two different Kafka Connectors: a source connector and a sink connector.


REMEMBER

To build a flow for running in Kafka Connect, build it within a process group using a traditional deployment of NiFi. Then export the process group by right-clicking it and selecting Download Flow or by saving the flow to a NiFi Registry instance.

Source connector

The NiFi source connector is responsible for obtaining data from one source and delivering that data to Kafka.


WARNING

Only route the data to an output port. Do not use processors such as PublishKafka to deliver directly to Kafka itself.

Any flowfile delivered to the output port is obtained by the connector and delivered to Kafka.

Each flowfile is delivered as a single Kafka record. In some situations, you may need to split the flowfile to avoid Kafka size limitations.

Sink connector

The NiFi sink connector is responsible for obtaining data from Kafka and delivering data to some other service.

WARNING

Only an input port should receive the data. The dataflow should not use processors such as ConsumeKafka to source data directly from Kafka itself.

Each Kafka record flowfile is delivered as a single flowfile. Depending on the destination, you may want to consolidate multiple Kafka records together before delivering them.

TIP

For example, if delivering to HDFS, you don't want to send each individual Kafka message as a separate file to prevent HDFS small file problems.

Cloudera DataFlow

Running NiFi on traditional bare metal or virtual machine (VM) infrastructure works great. However, it has some inherent issues:

- **Unless separated to different clusters, the tenants are all sharing the same resources.** This means they're subject to multi-tenancy related issues such as resource contention and failure domains.

  One way to overcome this limitation is to simply separate the different tenants to different clusters. This solution, however, comes with the additional overhead of managing and monitoring those clusters.

- **Properly sizing your cluster can be difficult.** Too small of a cluster and you won't meet your service level agreement (SLA). Sizing a cluster for peak periods solves that, but you might find yourself with an idle cluster off-peak.

  It's possible to manually scale up a NiFi cluster, but it's a multi-step process and requires you to constantly monitor the resource usage, and in time take care of deploying the software and setting up the required security (for example, certificates). Downscaling requires even more planning because it can potentially cause data loss if not done properly.

To address these challenges and others, Cloudera released DataFlow that enables running NiFi on Kubernetes based on a brand new Kubernetes operator, which encapsulates all the day-to-day knowledge required to manage a NiFi cluster.

- **Resource contention:** When running on Kubernetes, you can easily deploy each process group or multi-level nested process groups to their own NiFi clusters. This ensures that each process group gets a more accurate set of resources when compared to simply putting everything in one big monolith cluster.
- **Auto-scaling clusters:** When you deploy a flow, you can set lower and upper boundaries for the NiFi cluster to auto-scale in between. (This is in addition to a node profile that determines how many cores and RAM each node has.) This way, if you missed sizing estimates, the Kubernetes operator can readjust to the correct size as necessary. Auto-scaling occurs based on CPU utilization.
- **Central monitoring dashboard:** Cloudera provides an out-of-the-box dashboard for monitoring multiple clusters spread across multiple environments and multiple clouds. When you create a cluster, you define a set of key performance indicators (KPI) — for example, a certain queue's utilization or CPU utilization. That way, you automatically get notified when one of your KPIs is breached.
- **Simplifying upgrades:** When a maintenance release or hotfix is available for deployment, it's automatically available for NiFi administrators to deploy with a push of a button, allowing administrators to independently upgrade their deployments based on business needs and constraints. The deployment upgrade is carried out node by node in a rolling fashion, essentially preventing downtime.

Cloudera DataFlow Functions (DFF)

Serverless architectures are a way of software design where you can build applications without having to worry about managing infrastructure. They are becoming very attractive to consumers as they are easy to use, relatively inexpensive, and scale up very fast.

Function as a Service (FaaS) goes one step further than Software as a Service (SaaS) in reducing the complexity of building the architecture required for the application and instead offers you a way to directly invoke the function on-demand. These functions are typically triggered based on specific events and are best suited for burst workloads. They are offered by most of the cloud service providers (AWS Lambda, Azure Functions, and Google Cloud Functions are a few).

NiFi already provides a no-code UI to accelerate the development of your functions with an ever-growing set of processors and integrations to process your data and a robust software development life cycle solution around it. By using DataFlow Functions, you don't need to code your functions yourself anymore. Just design your flow in NiFi and you can be running in a few minutes while leveraging the 450+ processors already available.

Use cases of DataFlow Functions

There is a very wide range of use cases where DataFlow Functions is a great fit. The best source of inspiration is to look at the case studies of AWS Lambda, Azure Functions, and Google Cloud Functions. All these use cases can be implemented with DataFlow Functions as it provides a no-code UI to easily and quickly build any function.

Some of the possible use cases include:

- Real-time processing of files while they land into an object store
- Integrate with third-party services and APIs to expose microservices
- Process streams of data (IoT use cases, cybersecurity, fraud detection, for example)
- Integrate with mobile backends

Advantages of DataFlow Functions

When used for the right use cases, DataFlow Functions provides many advantages including:

- FaaS provides virtually unlimited (depending on the quotas and limitations of each cloud provider) scaling and it can be a very good option to handle bursts of events.

- There is no need to provision servers or Kubernetes clusters. Instead, FaaS uses the cloud native serverless framework to launch the infrastructure in the background.
- They are very cheap.
- There is no need to patch servers or do server maintenance.
- After the initial cold start, the subsequent runs complete very quickly — as fast as microseconds based on the function.

Chapter 8
Seven NiFi Resources

Here we present seven resources that provide more information to help you successfully use Apache NiFi:

- Apache NiFi docs `https://bit.ly/nifi-docs`
- Hello World sample code from Chapter 2 `https://github.com/drnice/NifiHelloWorld`
- Apache Technical Wiki `https://cwiki.apache.org/confluence/display/NIFI/FAQs`
- Cloudera DataFlow `http://bit.ly/nifi-for-dummies-cdf-page`
- Cloudera Flow Management Datasheet `http://bit.ly/nifi-for-dummies-cfm-datasheet`
- NiFi anti-patterns videos `http://bit.ly/nifi-anti-patterns`
- NiFi tutorials `http://bit.ly/cloudera-nifi-tutorials`

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

CLOUDERA

Go to Dummies.com®
for videos, step-by-step photos, how-to articles, or to shop!

ISBN: 978-1-119-81055-1
Not for resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.