Data Engineering with Python

Work with massive datasets to design data models and automate data pipelines using Python

Paul Crickard

Data Engineering with Python

Work with massive datasets to design data models and automate data pipelines using Python

Paul Crickard

BIRMINGHAM—MUMBAI

Data Engineering with Python

Commissioning Editor: Sunith Shetty

Acquisition Editor: Reshma Raman

Senior Editor: Roshan Kumar

Content Development Editor: Athikho Sapuni Rishana

Technical Editor: Manikandan Kurup

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Alishon Mendonca

First published: October 2020

Production reference: 1231020

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83921-418-9

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Paul Crickard is the author of *Leaflet.js Essentials* and co-author of *Mastering Geospatial Analysis with Python*, and is also the Chief Information Officer at the Second Judicial District Attorney's Office in Albuquerque, New Mexico.

With a master's degree in political science and a background in community and regional planning, he combines rigorous social science theory and techniques to technology projects. He has presented at the New Mexico Big Data and Analytics Summit and the ExperienceIT NM Conference. He has given talks on data to the New Mexico Big Data Working Group, Sandia National Labs, and the New Mexico Geographic Information Council.

About the reviewers

Stefan Marwah has enjoyed programming for over ten years, which led him to undertake a bachelor's degree in computer science from the reputable Monash University. During his time at the university, he built a mobile application that detected if an elderly person had Alzheimer's disease with help of natural language processing, speech recognition, and neural networks, which secured him an award from Microsoft. He has experience in both engineering and analytical roles that are rooted in his passion for leveraging data and artificial intelligence to make impactful decisions within different organizations. He currently works as a data engineer and also teaches part-time on topics around data science at Step Function Coaching.

Andre Sionek is a data engineer at Gousto, in London. He started his career by founding his own company, Polyteck, a free science and technology magazine for university students. But he only jumped into the world of data and analytics during an internship at the collections department of a Brazilian bank. He also worked with credit modeling for a large cosmetics group and for start-ups before moving to London. He regularly teaches data engineering courses, focusing on infrastructure as code and productionization. He also writes about data for his blog and competes on Kaggle sometimes.

Miles Obare is a software engineer at Microsoft in the Azure team. He is currently building tools that enable customers to migrate their server workloads to the cloud. He also builds real-time, scalable backend systems and data pipelines for enterprise customers. Formerly, he worked as a data engineer for a financial start-up, where his role involved developing and deploying data pipelines and machine learning models to production. His areas of expertise include distributed systems, computer architecture, and data engineering. He holds a bachelor's degree in electrical and computer engineering from Jomo Kenyatta University and contributes to open source projects in his free time.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

3
Reading and Writing Files

4
Working with Databases

5
Cleaning, Transforming, and Enriching Data

6
Building a 311 Data Pipeline

Section 2: Deploying Data Pipelines in Production

7
Features of a Production Pipeline

8
Version Control with the NiFi Registry

9
Monitoring Data Pipelines

10
Deploying Data Pipelines

11
Building a Production Data Pipeline

Section 3: Beyond Batch – Building Real-Time Data Pipelines

12
Building a Kafka Cluster

13
Streaming Data with Apache Kafka

14
Data Processing with Apache Spark

15
Real-Time Edge Data with MiNiFi, Kafka, and Spark

Appendix

Other Books You May Enjoy

Index

Preface

Data engineering provides the foundation for data science and analytics and constitutes an important aspect of all businesses. This book will help you to explore various tools and methods that are used to understand the data engineering process using Python.
The book will show you how to tackle challenges commonly faced in different aspects of data engineering. You'll start with an introduction to the basics of data engineering, along with the technologies and frameworks required to build data pipelines to work with large datasets. You'll learn how to transform and clean data and perform analytics to get the most out of your data. As you advance, you'll discover how to work with big data of varying complexity and production databases and build data pipelines. Using real-world examples, you'll build architectures on which you'll learn how to deploy data pipelines.

By the end of this Python book, you'll have gained a clear understanding of data modeling techniques, and will be able to confidently build data engineering pipelines for tracking data, running quality checks, and making necessary changes in production.

Who this book is for

This book is for data analysts, ETL developers, and anyone looking to get started with, or transition to, the field of data engineering or refresh their knowledge of data engineering using Python. This book will also be useful for students planning to build a career in data engineering or IT professionals preparing for a transition. No previous knowledge of data engineering is required.

What this book covers

Chapter 1, *What Is Data Engineering*, defines data engineering. It will introduce you to the skills, roles, and responsibilities of a data engineer. You will also learn how data engineering fits in with other disciplines, such as data science.

Chapter 2, *Building Our Data Engineering Infrastructure*, explains how to install and configure the tools used throughout this book. You will install two databases – ElasticSearch and PostgreSQL – as well as NiFi, Kibana, and, of course, Python.

Chapter 3, *Reading and Writing Files*, provides an introduction to reading and writing files in Python as well as data pipelines in NiFi. It will focus on **Comma Seperated Values** (**CSV**) and **JavaScript Object Notation** (**JSON**) files.

Chapter 4, *Working with Databases*, explains the basics of working with SQL and NoSQL databases. You will query both types of databases and view the results in Python and through the use of NiFi. You will also learn how to read a file and insert it into the databases.

Chapter 5, *Cleaning and Transforming Data*, explains how to take the files or database queries and perform basic exploratory data analysis. This analysis will allow you to view common data problems. You will then use Python and NiFi to clean and transform the data with a view to solving those common data problems.

Chapter 6, *Project – Building a 311 Data Pipeline*, sets out a project in which you will build a complete data pipeline. You will learn how to read from an API and use all of the skills acquired from previous chapters. You will clean and transform the data as well as enrich it with additional data. Lastly, you will insert the data into a warehouse and build a dashboard to visualize it.

Chapter 7, *Features of a Production Data Pipeline*, covers what is needed in a data pipeline to make it ready for production. You will learn about atomic transactions and how to make data pipelines idempotent.

Chapter 8, *Version Control Using the NiFi Registry*, explains how to version control your data pipelines. You will install and configure the NiFi registry. You will also learn how to configure the registry to use GitHub as the source of your NiFi processors.

Chapter 9, *Monitoring and Logging Data Pipelines*, teaches you the basics of monitoring and logging data pipelines. You will learn about the features of the NiFi GUI for monitoring. You will also learn how to use NiFi processors to log and monitor performance from within your data pipelines. Lastly, you will learn the basics of the NiFi API.

Chapter 10, *Deploying Your Data Pipelines*, proposes a method for building test and production environments for NiFi. You will learn how to move your completed and version-controlled data pipelines into a production environment.

Chapter 11, *Project – Building a Production Data Pipeline*, explains how to build a production data pipeline. You will use the project from *Chapter 6* and add a number of features. You will version control the data pipeline as well as adding monitoring and logging features.

Chapter 12, *Building an Apache Kafka Cluster*, explains how to install and configure a three-node Apache Kafka cluster. You will learn the basics of Kafka – streams, topics, and consumers.

Chapter 13, *Streaming Data with Kafka*, explains how, using Python, you can write to Kafka topics and how to consume that data. You will write Python code for both consumers and producers using a third-party Python library.

Chapter 14, *Data Processing with Apache Spark*, walks you through the installation and configuration of a three-node Apache Spark cluster. You will learn how to use Python to manipulate data in Spark. This will be reminiscent of working with pandas DataFrames from *Section 1* of this book.

Chapter 15, *Project – Real-Time Edge Data – Kafka, Spark, and MiNiFi*, introduces MiNiFi, which is a separate project to make NiFi available on low-resource devices such as Internet of Things devices. You will build a data pipeline that sends data from MiNiFi to your NiFi instance.

The *Appendix* teaches you the basics of clustering with Apache NiFi. You will learn how to distribute data pipelines and some caveats in doing so. You will also learn how to allow data pipelines to run on a single, specified node and not run distributed while in a cluster.

To get the most out of this book

You should have a basic understanding of Python. You will not be required to know any existing libraries, just a fundamental understanding of variables, functions, and how to run a program. You should also know the basics of Linux. If you can run a command in the terminal and open new terminal windows, that should be sufficient.

Software/hardware covered in the book	OS requirements
Python 3.x	Windows, macOS X, and Linux (any)
Spark 3.x	Linux
NiFi 1.x	Windows, macOS X, Linux
PostgreSQL 13.x	Windows, macOS X, Linux
ElasticSearch 7.x	Windows, macOS X, Linux
Kibana 7.x	Windows, macOS X, Linux
Apache Kafka 2.x	Linux, macOS X

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book at `https://github.com/PacktPublishing/Data-Engineering-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781839214189_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Next, pass the arguments dictionary to `DAG()`."

A block of code is set as follows:

```
import datetime as dt
from datetime import timedelta
```

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
```

```
import pandas as pd
```

Any command-line input or output is written as follows:

```
# web properties #
```

```
nifi.web.http.port=9300
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on **DAG** and select **Tree View**."

> **Tips or important notes**
>
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

Section 1: Building Data Pipelines – Extract Transform, and Load

This section will introduce you to the basics of data engineering. In this section, you will learn what data engineering is and how it relates to other similar fields, such as data science. You will cover the basics of working with files and databases in Python and using Apache NiFi. Once you are comfortable with moving data, you will be introduced to the skills required to clean and transform data. The section culminates with the building of a data pipeline to extract 311 data from SeeClickFix, transform it, and load it into another database. Lastly, you will learn the basics of building dashboards with Kibana to visualize the data you have loaded into your database.

This section comprises the following chapters:

- *Chapter 1*, *What is Data Engineering?*
- *Chapter 2*, *Building Our Data Engineering Infrastructure*
- *Chapter 3*, *Reading and Writing Files*
- *Chapter 4*, *Working with Databases*
- *Chapter 5*, *Cleaning and Transforming Data*
- *Chapter 6*, *Building a 311 Data Pipeline*

The graph view clearly shows the dependencies in the DAG and the order in which tasks will run. To watch the DAG run, switch back to **Tree View**. To the left of the DAG name, switch the DAG to **On**. Select **Trigger DAG** and you will be prompted whether you want to run it now. Select **Yes** and the page will refresh. I have run the DAG several times, and you can see the status of those runs in the following screenshot:

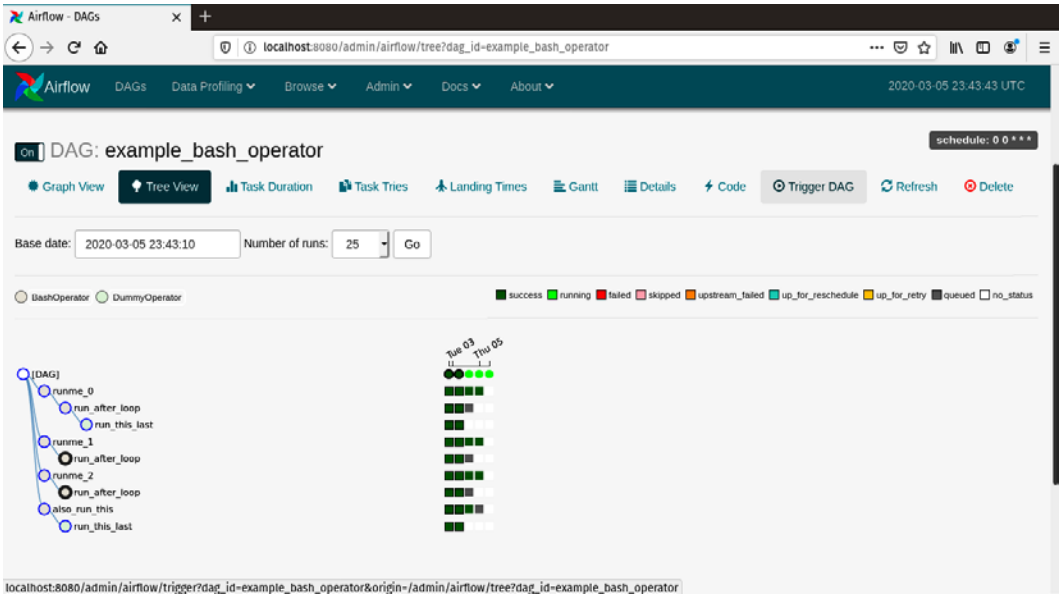

Figure 2.17 – Multiple runs of the execute_bash_operator DAG

Notice that there are two completed, successful runs of the DAG and three runs that are still running, with four queued tasks in those runs waiting. The examples are great for learning how to use the Airflow GUI, but they will be cluttered later. While this does not necessarily create a problem, it will be easier to find the tasks you created without all the extras.

You can remove the examples by editing the `airflow.cfg` file. Using `vi` or an editor of your choice, find the following line and change `True` to `False`:

```
load_examples = True
```

The `airflow.cfg` file is shown in the following screenshot, with the cursor at the line you need to edit:

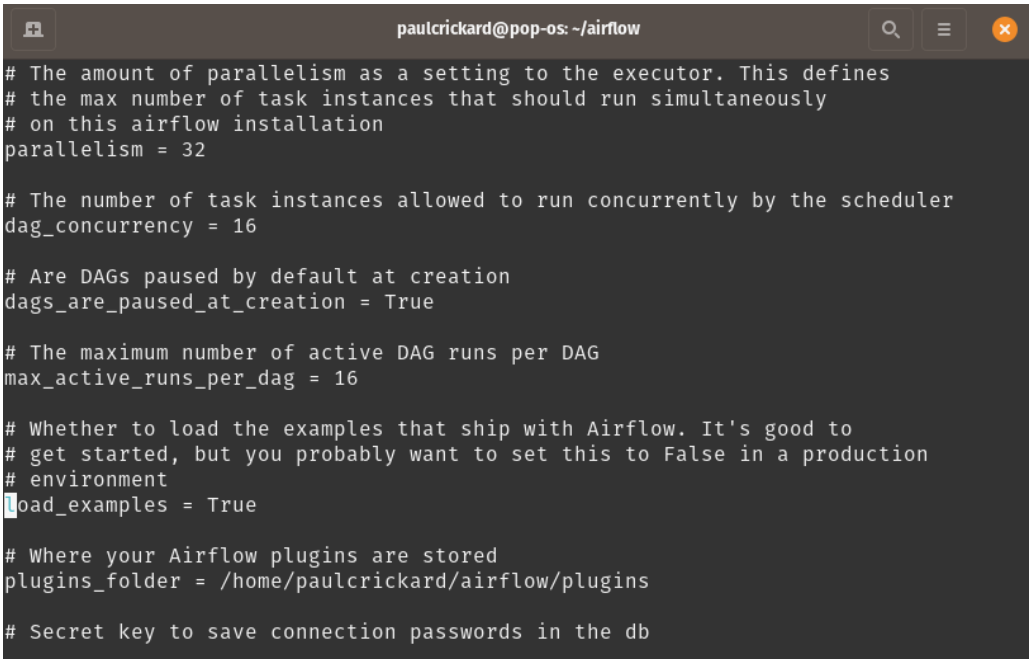

Figure 2.18 – Setting load_examples = False

Once you have edited the `airflow.cfg` file, you must shut down the web server. Once the web server has stopped, the changes to the configuration need to be loaded into the database. Remember that you set up the database earlier as the first step after `pip`, installing Airflow using the following command:

```
airflow initdb
```

To make changes to the database, which is what you want to do after changing the `airflow.cfg` file, you need to reset it. You can do that using the following snippet:

```
airflow resetdb
```

This will load in the changes from `airflow.cfg` to the metadata database. Now, you can restart the web server. When you open the GUI at `http://localhost:8080`, it should be empty, as shown in the following screenshot:

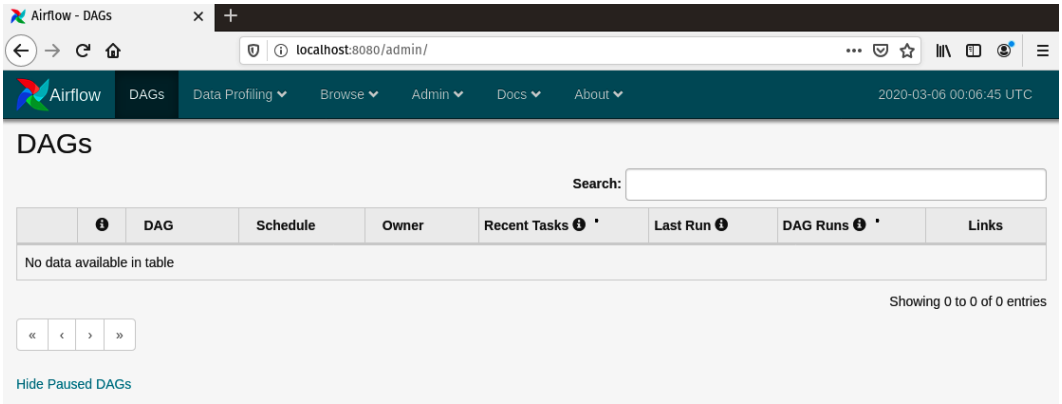

Figure 2.19 – Clean Airflow. Not a single DAG in sight

Airflow is clean and ready to load in the DAGs that you will create in the next chapter.

Installing and configuring Elasticsearch

Elasticsearch is a search engine. In this book, you will use it as a NoSQL database. You will move data both to and from Elasticsearch to other locations. To download Elasticsearch, take the following steps:

1. Use `curl` to download the files, as shown:

```
curl https://artifacts.elastic.co/downloads/
elasticsearch/elasticsearch-7.6.0-darwin-x86_64.tar.gz
--output elasticsearch.tar.gz
```

2. Extract the files using the following command:

```
tar xvzf elasticsearch.tar.gz
```

3. You can edit the `config/elasticsearch.yml` file to name your node and cluster. Later in this book, you will set up an Elasticsearch cluster with multiple nodes. For now, I have changed the following properties:

```
cluster.name: DataEngineeringWithPython
node.name: OnlyNode
```

4. Now, you can start Elasticsearch. To start Elasticsearch, run the following:

```
bin/elasticsearch
```

5. Once Elasticsearch has started, you can see the results at `http://localhost:9200`. You should see the following output:

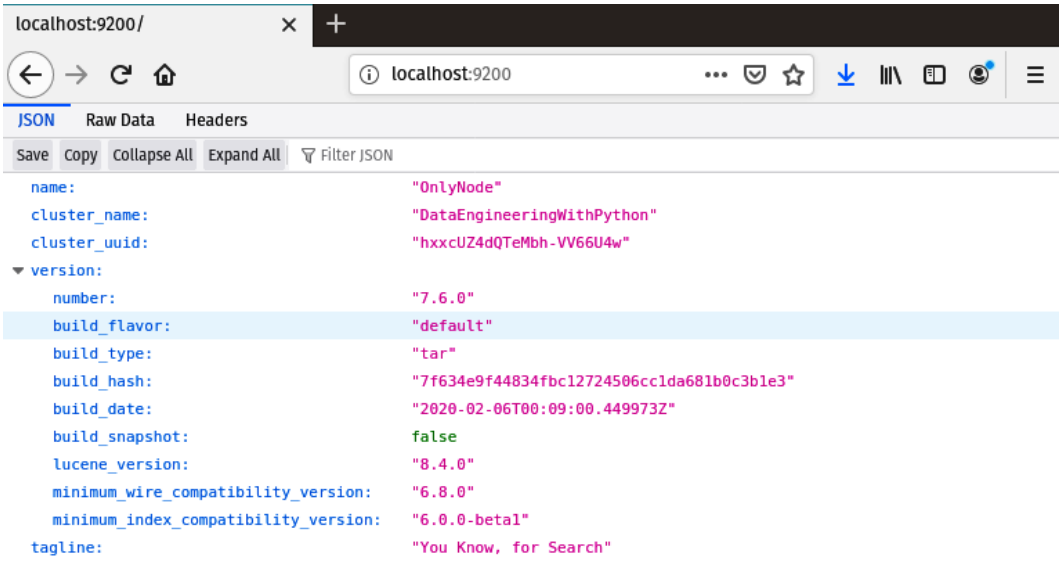

Figure 2.20 – Elasticsearch running

Now that you have a NoSQL database running, you will need a relational database as well.

Installing and configuring Kibana

Elasticsearch does not ship with a GUI, but rather an API. To add a GUI to Elasticsearch, you can use Kibana. By using Kibana, you can better manage and interact with Elasticsearch. Kibana will allow you to access the Elasticsearch API in a GUI, but more importantly, you can use it to build visualizations and dashboards of your data held in Elasticsearch. To install Kibana, take the following steps:

1. Using `wget`, add the key:

```
wget -qO - https://artifacts.elastic.co/GPG-KEY-
elasticsearch | sudo apt-key add -
```

2. Then, add the repository along with it:

```
echo "deb https://artifacts.elastic.co/packages/7.x/
apt stable main" | sudo tee -a /etc/apt/sources.list.d/
elastic-7.x.list
```

3. Lastly, update `apt` and install Kibana:

```
sudo apt-get update
sudo apt-get install kibana
```

4. The configuration files for Kibana are located in `etc/kibana` and the application is in `/usr/share/kibana/bin`. To launch Kibana, run the following:

```
bin/kibana
```

5. When Kibana is ready, browse to `http://localhost:5601`. Kibana will look for any instance of Elasticsearch running on `localhost` at port `9200`. This is where you installed Elasticsearch earlier, and also why you did not change the port in the configuration. When Kibana opens, you will be asked to choose between **Try our sample data** and **Explore on my own**, as shown:

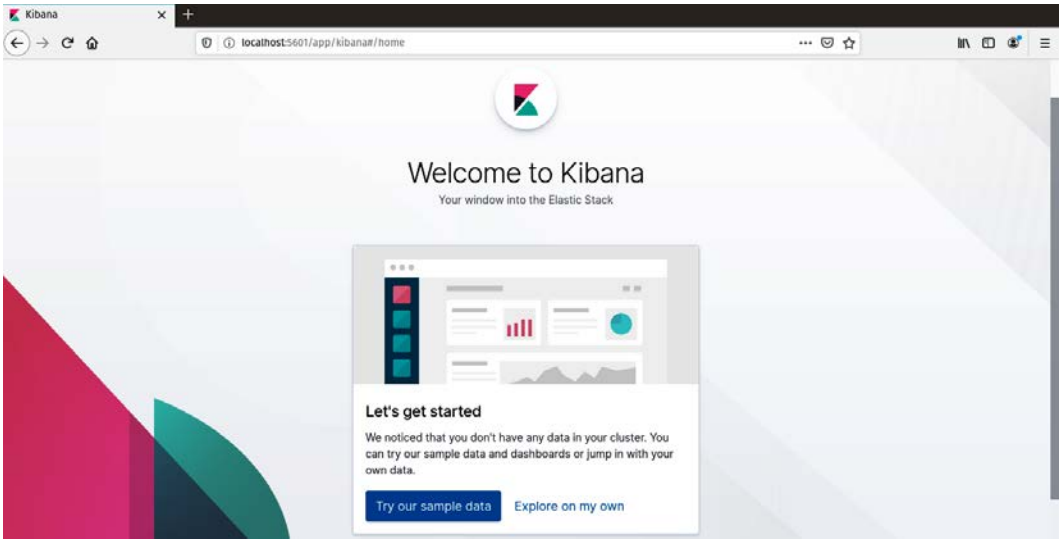

Figure 2.21 – First launch of Kibana

Explore on my own will take you to the main Kibana screen, but since you have not created an Elasticsearch index and have not loaded any data, the application will be blank.

To see the different tools available in Kibana, select **Try our sample data**, and choose the e-commerce data. The following screenshot shows the options for **Load our Sample Data**:

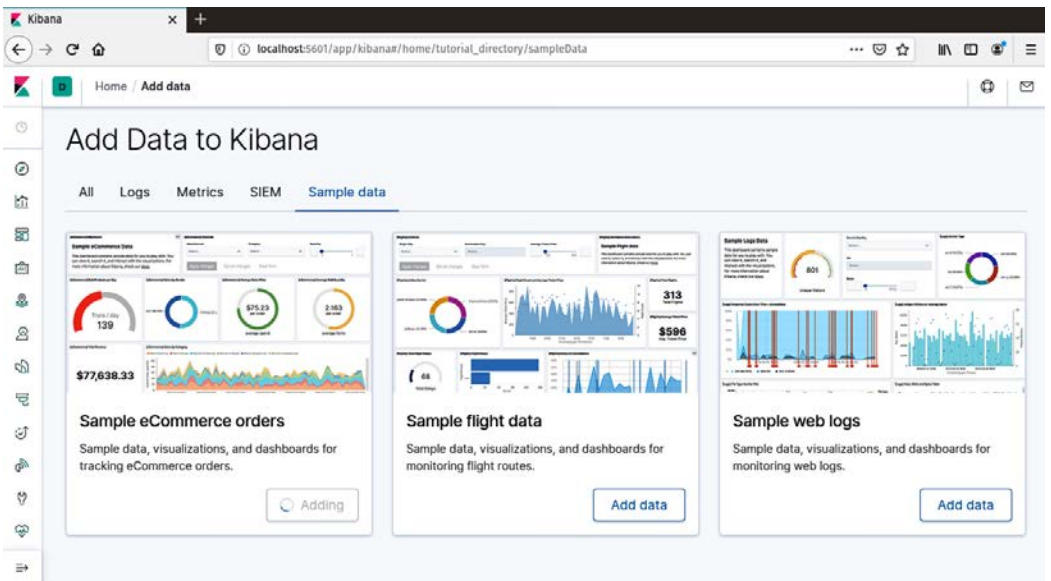

Figure 2.22 – Load sample data and visualizations

Once you have loaded the sample data, select the **Discover** icon. From the **Discover** section, you are able to look at records in the data. If there are dates, you will see a bar chart of counts on given time ranges. You can select a bar or change the date ranges from this tab. Selecting a record will show the data as a table or the JSON representation of the document. You can also run queries on the data from this tab and save them as objects to be used later in visualizations. The following screenshot shows the main **Discover** screen:

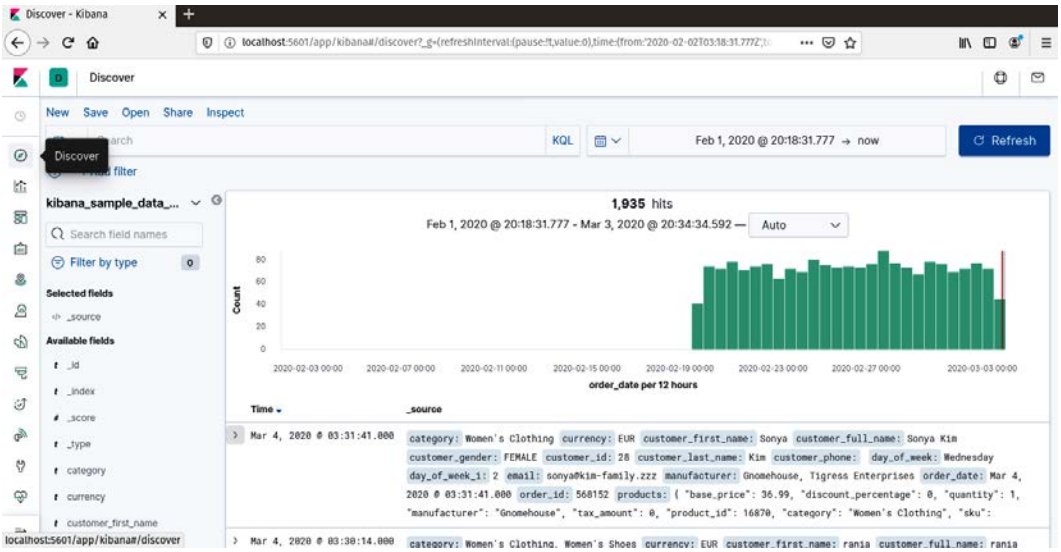

Figure 2.23 – The Discover tab

From the data available in the **Discover** tab or from a saved query, you can create visualizations. The visualizations include bar charts – horizontal and vertical, pie/donut charts, counts, markdown, heatmaps, and even a map widget to handle geospatial data. The e-commerce data contains geospatial data at the country level, but maps can also handle coordinates. The following screenshot shows a region map of the e-commerce data:

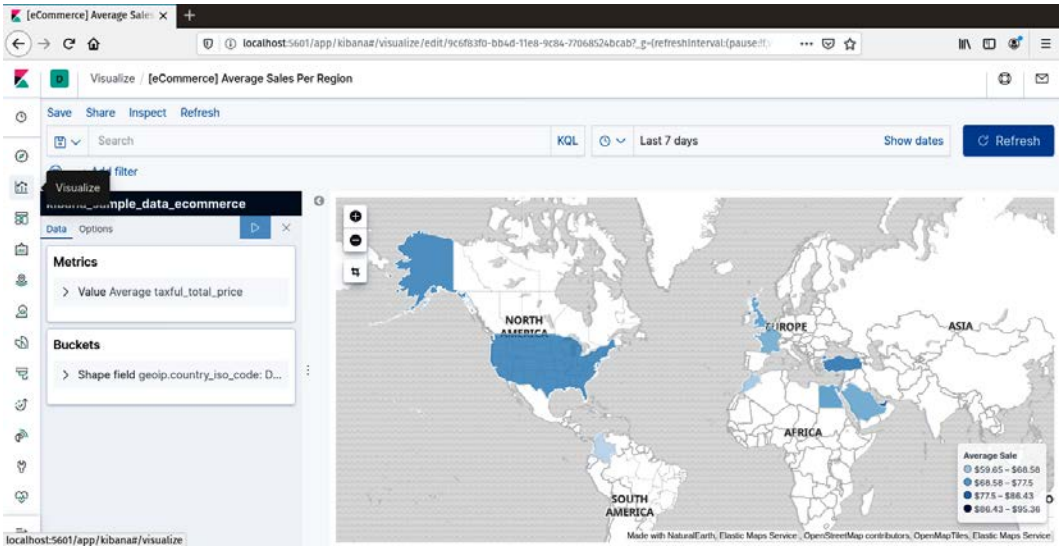

Figure 2.24 – A map visualization

When you have created several visualizations, from a single index or from multiple Elasticsearch indices, you can add them to a dashboard. Kibana allows you to load widgets using data from multiple indices. When you query or filter within the dashboard, as long as the field name exists in each of the indices, all of the widgets will update. The following screenshot shows a dashboard, made up of multiple visualizations of the e-commerce data:

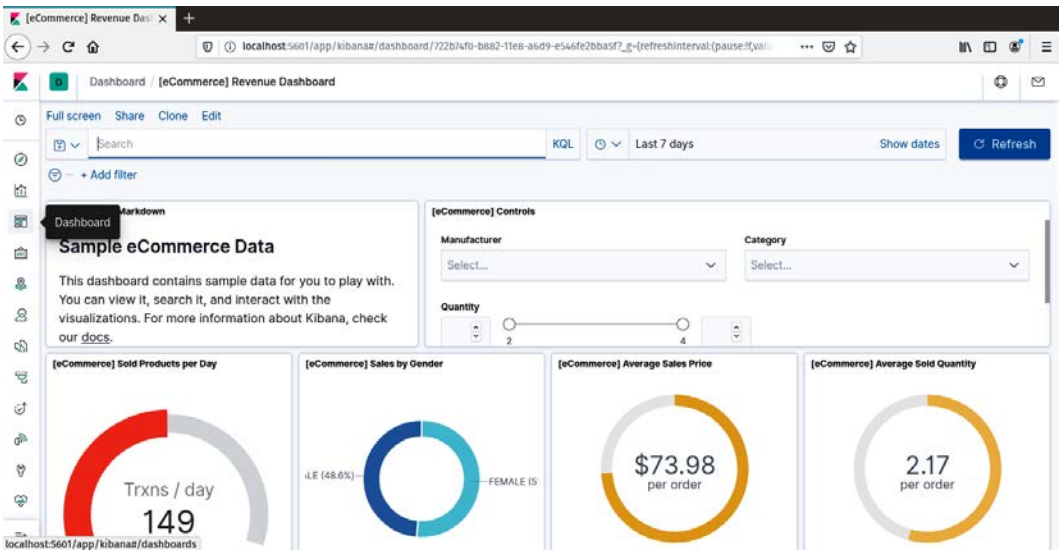

Figure 2.25 – A dashboard using multiple widgets from the e-commerce data

The **Developer Tools** tab comes in handy to quickly test Elasticsearch queries before you implement them in a data engineering pipeline. From this tab, you can create indices and data, execute queries to filter, search, or aggregate data. The results are displayed in the main window. The following screenshot shows a record being added to an index, then a search happening for a specific ID:

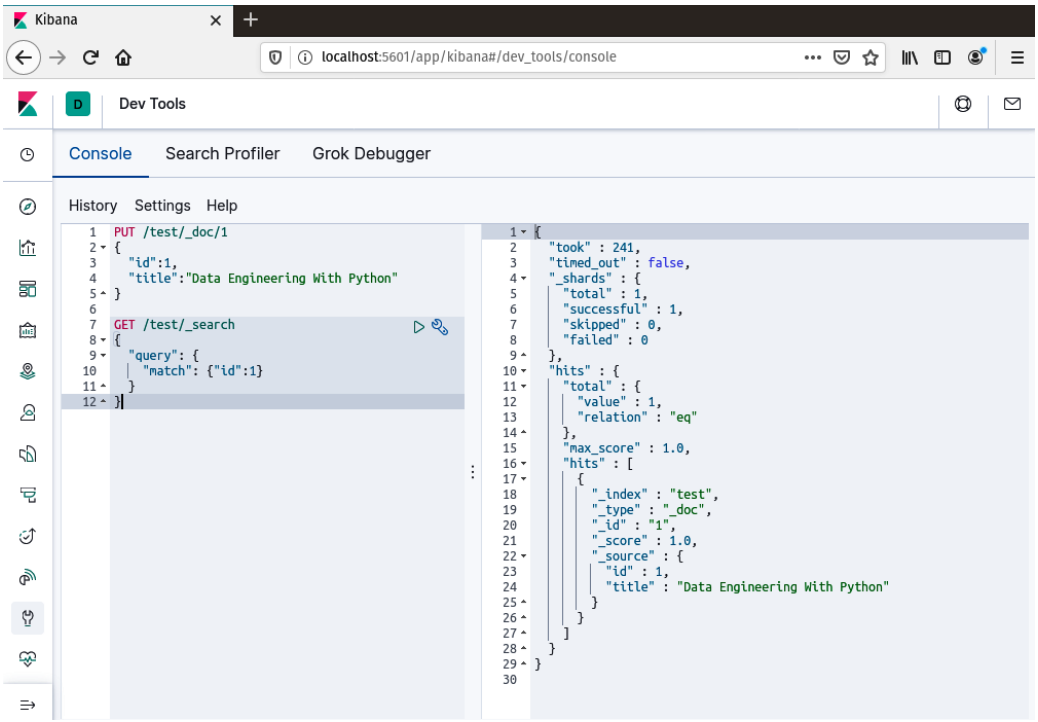

Figure 2.26 – A query on a single test record

Now that you have installed Elasticsearch and Kibana, the next two sections will walk you through installing PostgreSQL and pgAdmin 4. After that, you will have both a SQL and a NoSQL database to explore.

Installing and configuring PostgreSQL

PostgreSQL is an open source relational database. It compares to Oracle or Microsoft SQL Server. PostgreSQL also has a plugin – postGIS – which allows spatial capabilities in PostgreSQL. In this book, it will be the relational database of choice. PostgreSQL can be installed on Linux as a package:

1. For a Debian-based system, use `apt-get`, as shown:

```
sudo apt-get install postgresql-11
```

2. Once the packages have finished installing, you can start the database with the following:

```
sudo pg_ctlcluster 11 main start
```

3. The default user, `postgres`, does not have a password. To add one, connect to the default database:

```
sudo -u postgres psql
```

4. Once connected, you can alter the user and assign a password:

```
ALTER USER postgres PASSWORD ,postgres';
```

5. To create a database, you can enter the following command:

```
sudo -u postgres createdb dataengineering
```

Using the command line is fast, but sometimes, a GUI makes life easier. PostgreSQL has an administration tool – pgAdmin 4.

Installing pgAdmin 4

pgAdmin 4 will make managing PostgreSQL much easier if you are new to relational databases. The web-based GUI will allow you to view your data and allow you to visually create tables. To install pgAdmin 4, take the following steps:

1. You need to add the repository to Ubuntu. The following commands should be added to the repository:

```
wget --quiet -O - https://www.postgresql.org/media/keys/
ACCC4CF8.asc | sudo apt-key add -
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/
repos/apt/ `lsb_release -cs`-pgdg main" >> /etc/apt/
```

```
sources.list.d/pgdg.list'
sudo apt update
sudo apt install pgadmin4 pgadmin4-apache2 -y
```

2. You will be prompted to enter an email address for a username and then for a password. You should see the following screen:

Figure 2.27 – Creating a user for pgAdmin 4

3. When the install has completed, you can browse to `http://localhost/pgadmin4` and you will be presented with the login screen, as shown in the following screenshot. Enter the credentials for the user you just created during the install:

Figure 2.28 – Logging in to pgAdmin 4

Once you have logged in, you can manage your databases from the GUI. The next section will give you a brief tour of pgAdmin 4.

A tour of pgAdmin 4

After you log in to pgAdmin 4, you will see a dashboard with a server icon on the left side. There are currently no servers configured, so you will want to add the server you installed earlier in this chapter.

Click on the **Add new server** icon on the dashboard. You will see a pop-up window. Add the information for your PostgreSQL instance, as shown in the following screenshot:

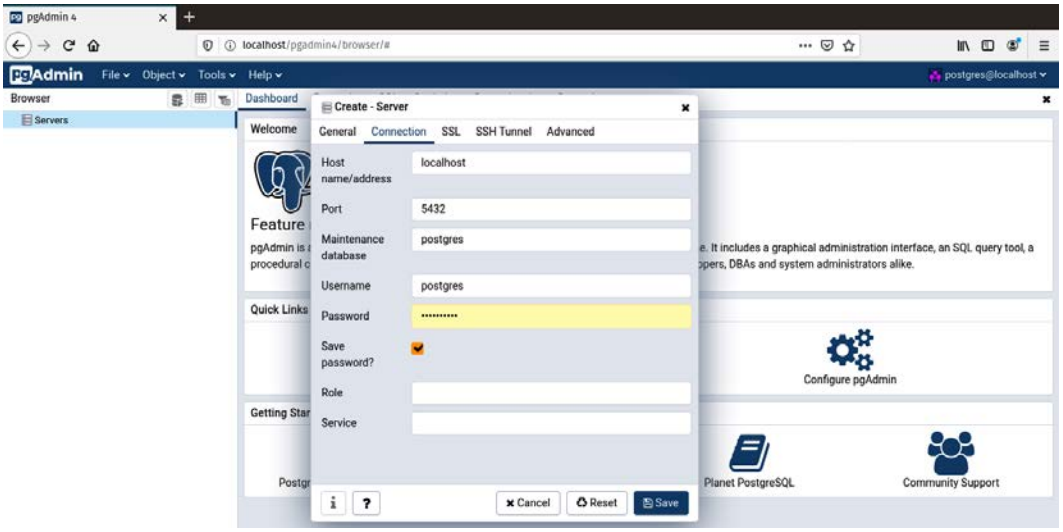

Figure 2.29 – Adding a new server

Once you add the server, you can expand the server icon and you should see the database you created earlier – `dataengineering`. Expand the `dataengineering` database, then `schemas`, then `public`. You will be able to right-click on **Tables** to add a table to the database, as shown in the following screenshot:

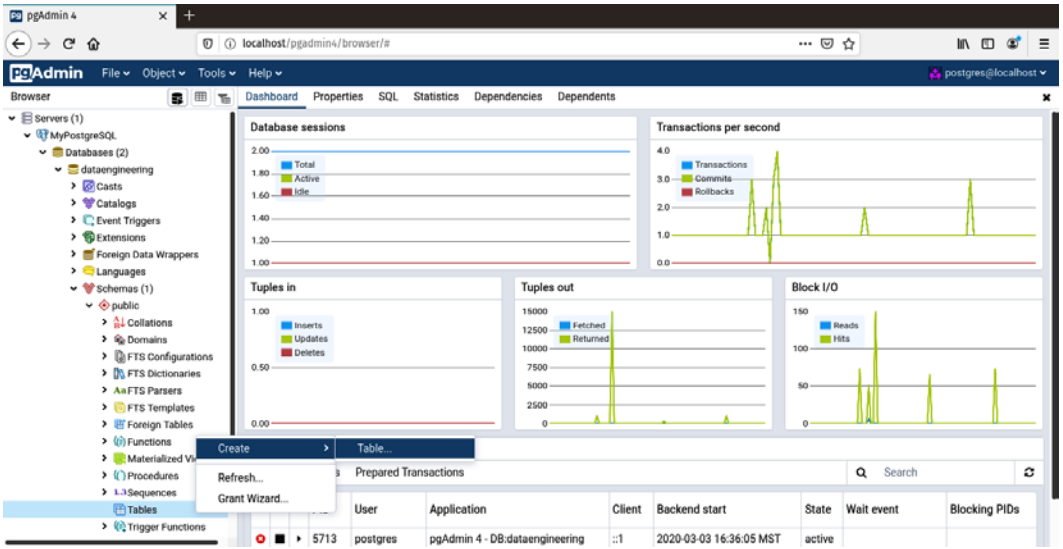

Figure 2.30 – Creating a table

To populate the table with data, name the table, then select the **Columns** tab. Create a table with some information about people. The table is shown in the following screenshot:

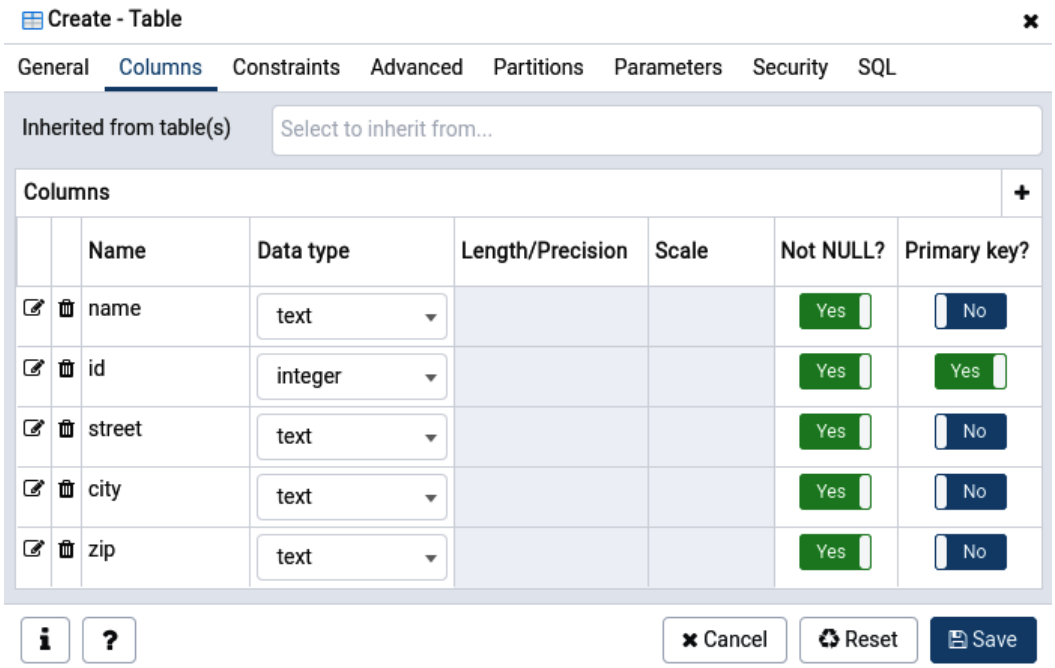

Figure 2.31 – Table data

In the next chapter, you will use Python to populate this table with data using the `faker` library.

Summary

In this chapter, you learned how to install and configure many of the tools used by data engineers. Having done so, you now have a working environment in which you can build data pipelines. In production, you would not run all these tools on a single machine, but for the next few chapters, this will help you learn and get started quickly. You now have two working databases – Elasticsearch and PostgreSQL – as well as two tools for building data pipelines – Apache NiFi and Apache Airflow.

In the next chapter, you will start to use Apache NiFi and Apache Airflow (Python) to connect to files, as well as Elasticsearch and PostgreSQL. You will build your first pipeline in NiFi and Airflow to move a CSV to a database.

3
Reading and Writing Files

In the previous chapter, we looked at how to install various tools, such as NiFi, Airflow, PostgreSQL, and Elasticsearch. In this chapter, you will be learning how to use these tools. One of the most basic tasks in data engineering is moving data from a text file to a database. In this chapter, you will read data from and write data to several different text-based formats, such as CSV and JSON.

In this chapter, we're going to cover the following main topics:

- Reading and writing files in Python
- Processing files in Airflow
- NiFi processors for handling files
- Reading and writing data to databases in Python
- Databases in Airflow
- Database processors in NiFi

Writing and reading files in Python

The title of this section may sound strange as you are probably used to seeing it written as reading and writing, but in this section, you will write data to files first, then read it. By writing it, you will understand the structure of the data and you will know what it is you are trying to read.

To write data, you will use a library named `faker`. `faker` allows you to easily create fake data for common fields. You can generate an address by simply calling `address()`, or a female name using `name_female()`. This will simplify the creation of fake data while at the same time making it more realistic.

To install `faker`, you can use `pip`:

```
pip3 install faker
```

With `faker` now installed, you are ready to start writing files. The next section will start with CSV files.

Writing and reading CSVs

The most common file type you will encounter is **Comma-Separated Values** (**CSV**). A CSV is a file made up of fields separated by commas. Because commas are fairly common in text, you need to be able to handle them in CSV files. This can be accomplished by using escape characters, usually a pair of quotes around text strings that could contain a comma that is not used to signify a new field. These quotes are called escape characters. The Python standard library for handling CSVs simplifies the process of handling CSV data.

Writing CSVs using the Python CSV Library

To write a CSV with the CSV library, you need to use the following steps:

1. Open a file in writing mode. To open a file, you need to specify a filename and a mode. The mode for writing is `w`, but you can also open a file for reading with `r`, appending with `a`, or reading and writing with `r+`. Lastly, if you are handling files that are not text, you can add `b`, for binary mode, to any of the preceding modes to write in bytes; for example, `wb` will allow you to write in bytes:

   ```
   output = open('myCSV.CSV',mode='w')
   ```

2. Create `CSV_writer`. At a minimum, you must specify a file to write to, but you can also pass additional parameters, such as a dialect. A dialect can be a defined CSV type, such as Excel, or it can be options such as the delimiter to use or the level of quoting. The defaults are usually what you will need; for example, the delimiter defaults to a comma (it is a CSV writer after all) and quoting defaults to `QUOTE_MINIMAL`, which will only add quotes when there are special characters or the delimiter within a field. So, you can create the writer as shown:

```
mywriter=csv.writer(output)
```

3. Include a header. You might be able to remember what the fields are in your CSV, but it is best to include a header. Writing a header is the same as writing any other row: define the values, then you will use `writerow()`, as shown:

```
header=['name','age']
mywriter.writerow(header)
```

4. Write the data to a file. You can now write a data row by using `writerow(0)` and passing some data, as shown:

```
data=['Bob Smith',40]
mywriter.writerow(data)
output.close()
```

Now, if you look in the directory, you will have a CSV file named `myCSV.CSV` and the contents should look as in the following screenshot:

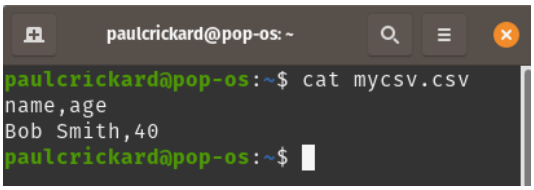

Figure 3.1 – The contents of mycsv.csv

Notice that when you used `cat` to view the file, the newlines were added. By default, `CSV_writer` uses a return and a newline (`'\r\n'`).

The preceding example was very basic. However, if you are trying to write a lot of data, you would most likely want to loop through some condition or iterate through existing data. In the following example, you will use `Faker` to generate 1,000 records:

```
from faker import Faker
import csv
output=open('data.CSV','w')
fake=Faker()
header=['name','age','street','city','state','zip','lng','lat']
mywriter=csv.writer(output)
mywriter.writerow(header)
for r in range(1000):
    mywriter.writerow([fake.name(),fake.random_int(min=18,
    max=80, step=1), fake.street_address(), fake.city(),fake.
    state(),fake.zipcode(),fake.longitude(),fake.latitude()])
    output.close()
```

You should now have a `data.CSV` file with 1,000 rows of names and ages.

Now that you have written a CSV, the next section will walk you through reading it using Python.

Reading CSVs

Reading a CSV is somewhat similar to writing one. The same steps are followed with slight modifications:

1. Open a file using `with`. Using `with` has some additional benefits, but for now, the one you will reap is not having to use `close()` on the file. If you do not specify a mode, `open` defaults to read (`r`). After `open`, you will need to specify what to refer to the file as; in this case, you will open the `data.CSV` file and refer to it as `f`:

   ```
   with open('data.csv') as f:
   ```

2. Create the reader. Instead of just using `reader()`, you will use `DictReader()`. By using the dictionary reader, you will be able to call fields in the data by name instead of position. For example, instead of calling the first item in a row as `row[0]`, you can now call it as `row['name']`. Just like the writer, the defaults are usually sufficient, and you will only need to specify a file to read. The following code opens `data.CSV` using the `f` variable name:

```
myreader=CSV.DictReader(f)
```

3. Grab the headers by reading a single line with `next()`:

```
headers=next(myreader)
```

4. Now, you can iterate through the rest of the rows using the following:

```
for row in myreader:
```

5. Lastly, you can print the names using the following:

```
    print(row['name'])
```

You should only see the 1,000 names scroll by. Now you have a Python dictionary that you can manipulate any way you need. There is another way to handle CSV data in Python and that requires `pandas`.

Reading and writing CSVs using pandas DataFrames

`pandas` DataFrames are a powerful tool not only for reading and writing data but also for the querying and manipulation of data. It does require a larger overhead than the built-in CSV library, but there are times when it may be worth the trade-off. You may already have `pandas` installed, depending on your Python environment, but if you do not, you can install it with the following:

```
pip3 install pandas
```

You can think of a `pandas` DataFrame as an Excel sheet or a table. You will have rows, columns, and an index. To load CSV data into a DataFrame, the following steps must be followed:

1. Import `pandas` (usually as `pd`):

```
import pandas as pd
```

2. Then, read the file using `read_csv()`. The `read_csv()` method takes several optional parameters, and one required parameter – the file or file-like buffer. The two optional parameters that may be of interest are `header`, which by defaultattempts to infer the headers. If you set `header=0`, then you can use the `names` parameter with an array of column names. If you have a large file and you just want to look at a piece of it, you can use `nrows` to specify the number of rows to read, so `nrows=100` means it will only read 100 rows for the data. In the following snippet, you will load the entire file using the defaults:

```
df=pd.read_csv()('data.CSV')
```

3. Let's now look at the first 10 records by using the following:

```
df.head(10)
```

Because you used `Faker` to generate data, you will have the same schema as in the following screenshot, but will have different values:

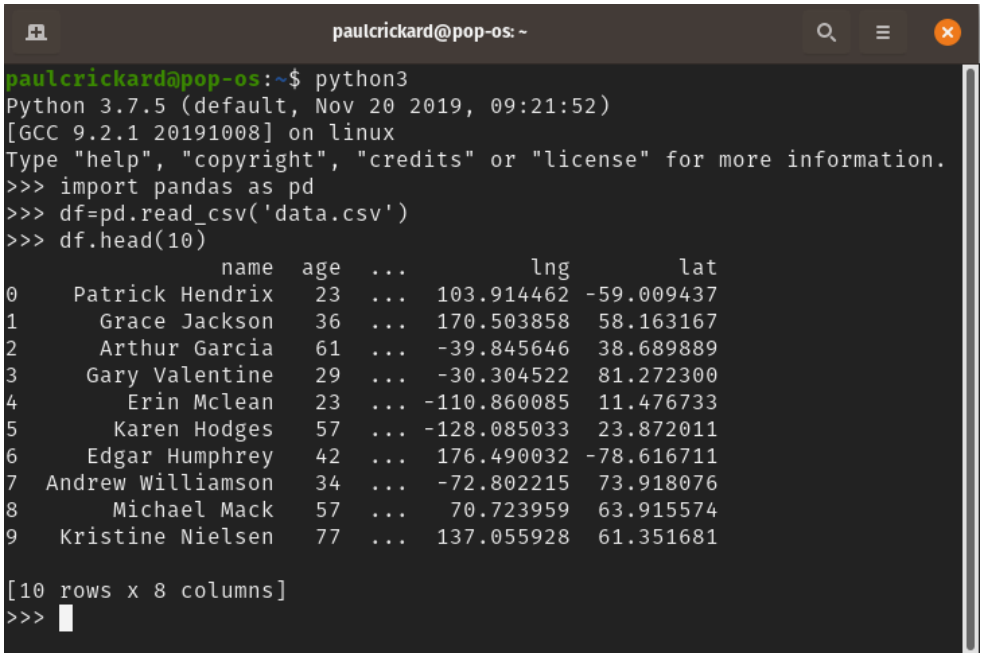

Figure 3.2 – Reading a CSV into a DataFrame and printing head()

You can create a DataFrame in Python with the following steps:

1. Create a dictionary of data. A dictionary is a data structure that stores data as a key:value pair. The value can be of any Python data type – for example, an array. Dictionaries have methods for finding `keys()`, `values()`, and `items()`. They also allow you to find the value of a key by using the key name in brackets – for example, `dictionary['key']` will return the value for that key:

```
data={'Name':['Paul','Bob','Susan','Yolanda'],
'Age':[23,45,18,21]}
```

2. Pass the data to the DataFrame:

```
df=pd.DataFrame(data)
```

3. The columns are specified as the keys in the dictionary. Now that you have a DataFrame, you can write the contents to a CSV using `to_csv()` and passing a filename. In the example, we did not set an index, which means the row names will be a number from 0 to *n*, where *n* is the length of the DataFrame. When you export to CSV, these values will be written to the file, but the column name will be blank. So, in a case where you do not need the row names or index to be written to the file, pass the `index` parameter to `to_csv()`, as shown:

```
df.to_csv('fromdf.CSV',index=False)
```

You will now have a CSV file with the contents of the DataFrame. How we can use the contents of this DataFrame for executing SQL queries will be covered in the next chapter. They will become an important tool in your toolbox and the rest of the book will lean on them heavily.

For now, let's move on to the next section, where you will learn about another common text format – **JSON**.

Writing JSON with Python

Another common data format you will probably deal with is **JavaScript Object Notation** (**JSON**). You will see JSON most often when making calls to **Application Programming Interfaces** (**APIs**); however, it can exist as a file as well. How you handle the data is very similar no matter whether you read it from a file or an API. Python, as you learned with CSV, has a standard library for handling JSON data, not surprisingly named `JSON-JSON`.

To write JSON using Python and the standard library, the following steps need to be observed:

1. Import the library and open the file you will write to. You also create the `Faker` object:

```
from faker import Faker
import json
output=open('data.JSON','w')
fake=Faker()
```

2. We will create 1,000 records, just as we did in the CSV example, so you will need to create a dictionary to hold the data. As mentioned earlier, the value of a key can be any Python data type – including an array of values. After creating the dictionary to hold the records, add a `'records'` key and initialize it with a blank array, as shown:

```
alldata={}
alldata['records']=[]
```

3. To write the records, you use `Faker` to create a dictionary, then append it to the array:

```
for x in range(1000):
      data={"name":fake.name(),"age":fake.random_int
            (min=18, max=80, step=1),
            "street":fake.street_address(),
            "city":fake.city(),"state":fake.state(),
            "zip":fake.zipcode(),
            "lng":float(fake.longitude()),
            "lat":float(fake.latitude())}
      alldata['records'].append(data)
```

4. Lastly, to write the JSON to a file, use the `JSON.dump()` method. Pass the data that you want to write and a file to write to:

```
json.dump(alldata,output)
```

You now have a `data.JSON` file that has an array with 1,000 records. You can read this file by taking the following steps:

1. Open the file using the following:

```
with open("data.JSON","r") as f:
```

2. Use `JSON.load()` and pass the file reference to the method:

```
data=json.load(f)
```

3. Inspect the json by looking at the first record using the following:

```
data['records'][0]
```

   Or just use the name:

```
data['records'][0]['name']
```

When you **load** and **dump** JSON, make sure you do not add an *s* at the end of the JSON terms. `loads` and `dumps` are different than `load` and `dump`. Both are valid methods of the JSON library. The difference is that `loads` and `dumps` are for strings – they do not serialize the JSON.

pandas DataFrames

Reading and writing JSON with DataFrames is similar to what we did with CSV. The only difference is that you change `to_csv` to `to_json()` and `read_csv()` to `read_json()`.

If you have a clean, well-formatted JSON file, you can read it using the following code:

```
df=pd.read_json('data.JSON')
```

In the case of the `data.JSON` file, the records are nested in a `records` dictionary. So, loading the JSON is not as straightforward as the preceding code. You will need a few extra steps, which are as follows. To load JSON data from the file, do the following:

1. Use the `pandas JSON` library:

```
import pandas.io.json as pd_JSON
```

2. Open the file and load it with the `pandas` version of `JSON.loads()`:

```
f=open('data.JSON','r')
data=pd_JSON.loads(f.read())
```

3. To create the DataFrame, you need to normalize the JSON. Normalizing is how you can flatten the JSON to fit in a table. In this case, you want to grab the individual JSON records held in the `records` dictionary. Pass that path – `records` – to the `record_path` parameter of `json_normalize()`:

```
df=pd_JSON.json_normalize(data,record_path='records')
```

You will now have a DataFrame that contains all the records in the `data.JSON` file. You can now write them back to JSON, or CSV, using DataFrames.

When writing to JSON, you can pass the `orient` parameter, which determines the format of the JSON that is returned. The default is columns, which for the `data.JSON` file you created in the previous section would look like the following data:

```
>>> df.head(2).to_json()
'{"name":{"0":"Henry Lee","1":"Corey Combs DDS"},"
age":{"0":42,"1":43},"street":{"0":"57850 Zachary
Camp","1":"60066 Ruiz Plaza Apt. 752"},"city":{"0":"Lake
Jonathon","1":"East Kaitlin"},"state":{"0":"Rhode Island","
1":"Alabama"},"zip":{"0":"93363","1":"16297"},"lng":{"0":-
161.561209,"1":123.894456},"lat":
{"0":-72.086145,"1":-50.211986}}'
```

By changing the `orient` value to `records`, you get each row as a record in the JSON, as shown:

```
>>> df.head(2).to_JSON(orient='records')
'[{"name":"Henry Lee","age":42,"street":"57850, Zachary
Camp","city":"Lake Jonathon","state":"Rhode Island",
"zip":"93363","lng":-161.561209,"lat":72.086145},{"name":"
Corey Combs DDS","age":43,"street":"60066 Ruiz Plaza Apt.
752","city":"EastKaitlin","state":"Alabama",
"zip":"16297","lng":123.894456, "lat":-50.211986}]'
```

I find that working with JSON that is oriented around `records` makes processing it in tools such as Airflow much easier than JSON in other formats, such as split, index, columns, values, or table. Now that you know how to handle CSV and JSON files in Python, it is time to learn how to combine tasks into a data pipeline using Airflow and NiFi. In the next section, you will learn how to build pipelines in Apache Airflow.

Building data pipelines in Apache Airflow

Apache Airflow uses Python functions, as well as Bash or other operators, to create tasks that can be combined into a **Directed Acyclic Graph** (**DAG**) – meaning each task moves in one direction when completed. Airflow allows you to combine Python functions to create tasks. You can specify the order in which the tasks will run, and which tasks depend on others. This order and dependency are what make it a DAG. Then, you can schedule your DAG in Airflow to specify when, and how frequently, your DAG should run. Using the Airflow GUI, you can monitor and manage your DAG. By using what you learned in the preceding sections, you will now make a data pipeline in Airflow.

Building a CSV to a JSON data pipeline

Starting with a simple DAG will help you understand how Airflow works and will help you to add more functions to build a better data pipeline. The DAG you build will print out a message using Bash, then read the CSV and print a list of all the names. The following steps will walk you through building the data pipeline:

1. Open a new file using the Python IDE or any text editor. Import the required libraries, as shown:

```
import datetime as dt
from datetime import timedelta

from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import
PythonOperator

import pandas as pd
```

   The first two imports bring in `datetime` and `timedelta`. These libraries are used for scheduling the DAG. The three Airflow imports bring in the required libraries for building the DAG and using the Bash and Python operators. These are the operators you will use to build tasks. Lastly, you import `pandas` so that you can easily convert between CSV and JSON.

2. Next, write a function to read a CSV file and print out the names. By combining the steps for reading CSV data and writing JSON data from the previous sections, you can create a function that reads in the `data.CSV` file and writes it out to JSON, as shown in the following code:

```
def CSVToJson():
    df=pd.read_CSV('/home/paulcrickard/data.CSV')
    for i,r in df.iterrows():
        print(r['name'])
    df.to_JSON('fromAirflow.JSON',orient='records')
```

   This function opens the file in a DataFrame. Then, it iterates through the rows, printing only the names, and lastly, it writes the CSV to a JSON file.

3. Now, you need to implement the Airflow portion of the pipeline. Specify the arguments that will be passed to `DAG()`. In this book, you will use a minimal set of parameters. The arguments in this example assign an owner, a start date, the number of retries in the event of a failure, and how long to wait before retrying. They are shown in the following dictionary:

```
default_args = {
    'owner': 'paulcrickard',
    'start_date': dt.datetime(2020, 3, 18),
    'retries': 1,
    'retry_delay': dt.timedelta(minutes=5),
}
```

4. Next, pass the arguments dictionary to `DAG()`. Create the DAG ID, which is set to `MyCSVDAG`, the dictionary of arguments (the `default_args` variable in the preceding code), and the schedule interval (how often to run the data pipeline). The schedule interval can be set using `timedelts`, or you can use a crontab format with the following presets or crontab:

   a) `@once`

   b) `@hourly` – `0 * * * *`

   c) `@daily` – `0 0 * * *`

   d) `@weekly` – `0 0 * * 0`

   e) `@monthly` – `0 0 1 * *`

   f) `@yearly` – `0 0 1 1 *`

crontab uses the format minute, hour, day of month, month, day of week. The value for `@yearly` is `0 0 1 1 *`, which means run yearly on January 1 (`1 1`), at 0:0 (midnight), on any day of the week (`*`).

> **Scheduling a DAG warning**
>
> The `start_date` variable of a DAG is `start_date` + the `schedule_interval`. This means if you schedule a DAG with a `start_date` value of today, and a `schedule_interval` value of daily, the DAG will not run until tomorrow.

The DAG is created with the following code:

```
with DAG('MyCSVDAG',
         default_args=default_args,
         schedule_interval=timedelta(minutes=5),
         # '0 * * * *',
         ) as dag:
```

5. You can now create your tasks using operators. Airflow has several prebuilt operators. You can view them all in the documentation at `https://airflow.apache.org/docs/stable/_api/airflow/operators/index.html`. In this book, you will mostly use the Bash, Python, and Postgres operators. The operators allow you to remove most of the boilerplate code that is required to perform common tasks. In the following snippet, you will create two tasks using the Bash and Python operators:

```
    print_starting = BashOperator(task_id='starting',
                      bash_command='echo "I am reading the
                      CSV now....."')

    CSVJson = PythonOperator(task_id='convertCSVtoJson',
                             python_callable=CSVToJson)
```

The preceding snippet creates a task using the `BashOperator` operator, which prints out a statement to let you know it is running. This task serves no purpose other than to allow you to see how to connect multiple tasks together. The next task, `CSVJson`, uses the `PythonOperator` operator to call the function you defined at the beginning of the file (`CSVToJson()`). The function reads the `data.CSV` file and prints the `name` field in every row.

6. With the tasks defined, you now need to make the connections between the tasks. You can do this using the `set_upstream()` and `set_downstream()` methods or with the bit shift operator. By using upstream and downstream, you can make the graph go from the Bash task to the Python task using either of two snippets; the following is the first snippet:

```
print_starting .set_downstream(CSVJson)
```

The following is the second snippet:

```
CSVJson.set_upstream(print_starting)
```

Using the bit shift operator, you can do the same; the following is the first option:

```
print_starting >>  CSVJson
```

The following is the second option:

```
CSVJson << print_starting
```

> **Note**
> Which method you choose is up to you; however, you should be consistent. In this book, you will see the bit shift operator setting the downstream.

7. To use Airflow and Scheduler in the GUI, you first need to make a directory for your DAGs. During the install and configuration of Apache Airflow, in the previous chapter, we removed the samples and so the DAG directory is missing. If you look at `airflow.cfg`, you will see the setting for `dags_folder`. It is in the format of `$AIRFLOW_HOME/dags`. On my machine, `$AIRFLOW_HOME` is `home/paulcrickard/airflow`. This is the directory in which you will make the `dags folder.e` configuration file showing where the folder should be.
8. Copy your DAG code to the folder, then run the following:

```
airflow webserver
airflow scheduler
```

9. Launch the GUI by opening your web browser and going to `http://localhost:8080`. You will see your DAG, as shown in the following screenshot:

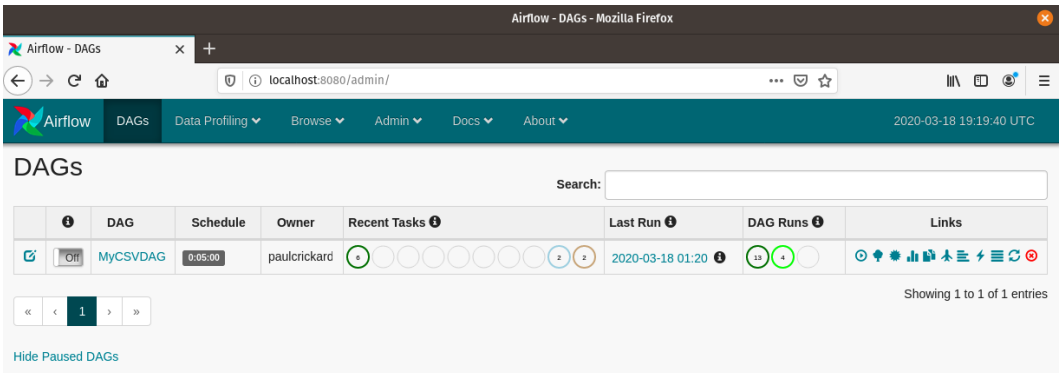

Figure 3.3 – The main screen of the Airflow GUI showing MyCSVDAG

10. Click on **DAGs** and select **Tree View**. Turn the DAG on, and then click **Go**. As the tasks start running, you will see the status of each run, as shown in the following screenshot:

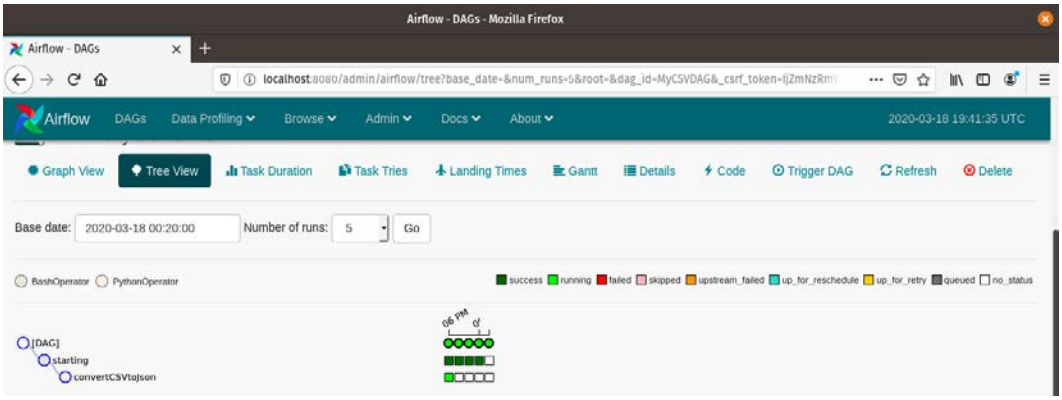

Figure 3.4 – Multiple runs of the DAG and the status of each task

11. You will see that there have been successful runs – each task ran and did so successfully. But there is no output or results. To see the results, click on one of the completed squares, as shown in the following screenshot:

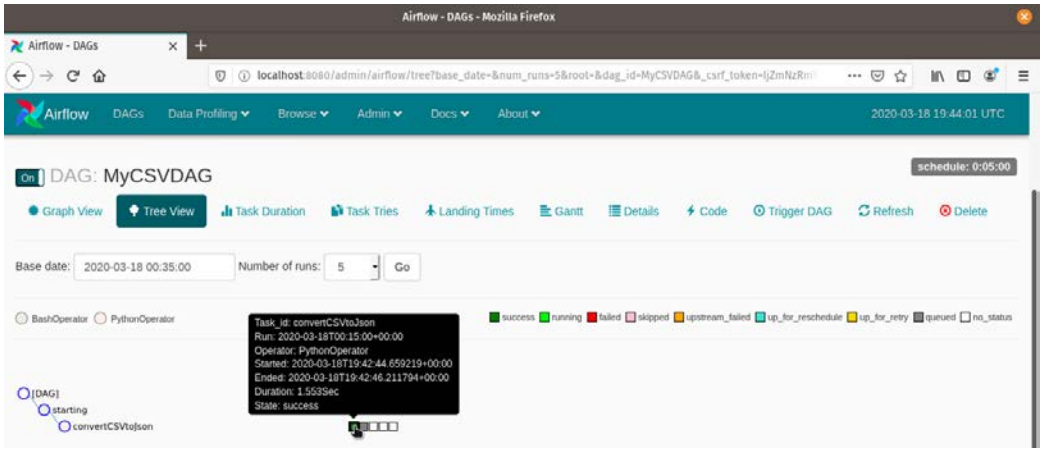

Figure 3.5 – Checking results by hovering over the completed task

12. You will see a popup with several options. Click the **View Log** button, as shown in the following screenshot:

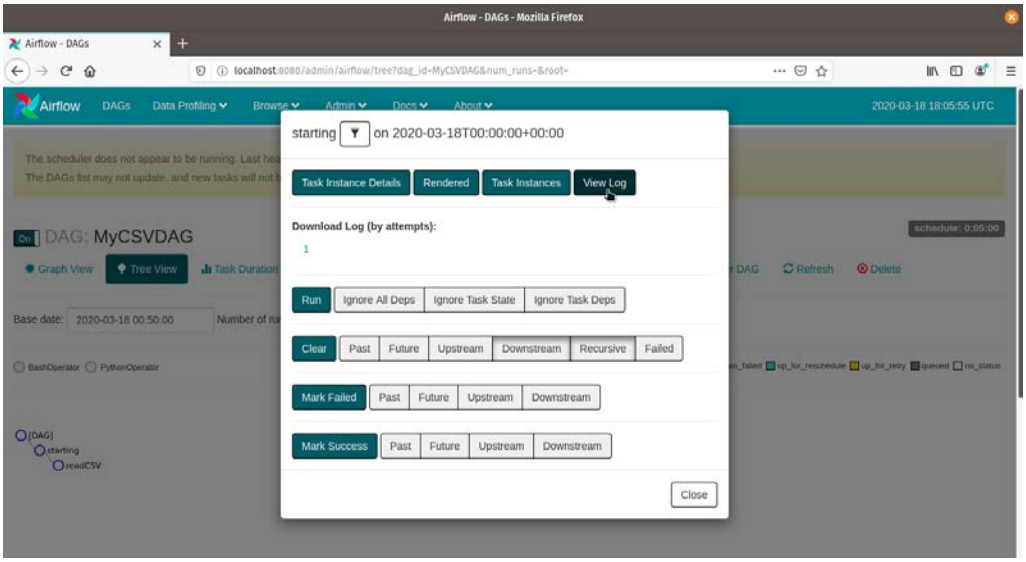

Figure 3.6 – Selecting View Log to see what happened in your task

13. You will be redirected to the log screen for the task. Looking at a successful run of the CSV task, you should see a log file similar to the one in the following screenshot:

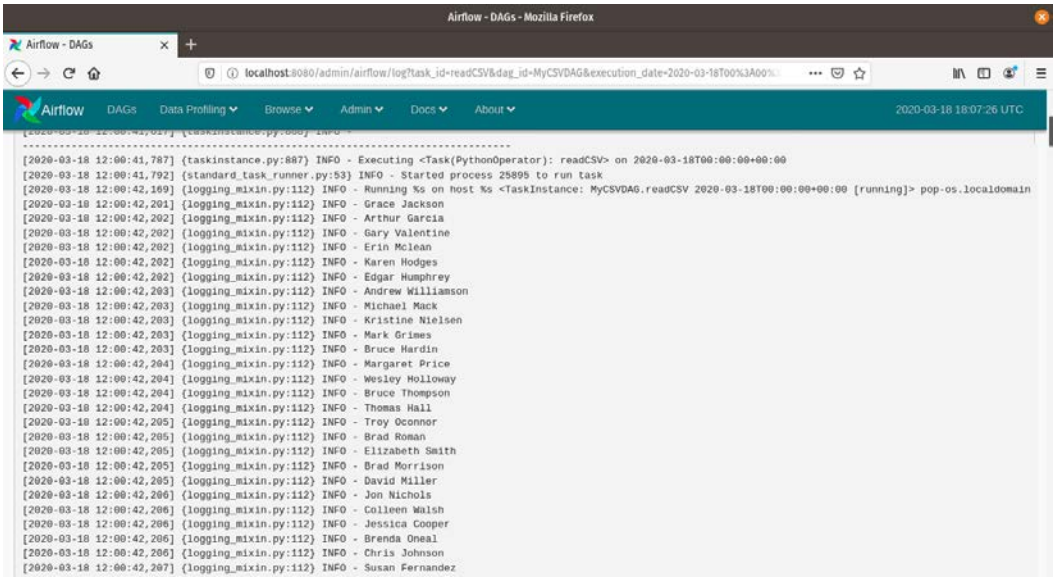

Figure 3.7 – Log of the Python task showing the names being printed

Congratulations! You have built a data pipeline with Python and ran it in Airflow. The result of your pipeline is a JSON file in your `dags` directory that was created from your `data.CSV` file. You can leave it running and it will continue to run at the specified `schedule_interval` time. Building more advanced pipelines will only require you to write more functions and connect them with the same process. But before you move on to more advanced techniques, you will need to learn how to use Apache NiFi to build data pipelines.

Handling files using NiFi processors

In the previous sections, you learned how to read and write CSV and JSON files using Python. Reading files is such a common task that tools such as NiFi have prebuilt processors to handle it. In this section, you will learn how to handle files using NiFi processors.

Working with CSV in NiFi

Working with files in NiFi requires many more steps than you had to use when doing the same tasks in Python. There are benefits to using more steps and using Nifi, including that someone who does not know code can look at your data pipeline and understand what it is you are doing. You may even find it easier to remember what it is you were trying to do when you come back to your pipeline in the future. Also, changes to the data pipeline do not require refactoring a lot of code; rather, you can reorder processors via drag and drop.

In this section, you will create a data pipeline that reads in the `data.CSV` file you created in Python. It will run a query for people over the age of 40, then write out that record to a file.

The result of this section is shown in the following screenshot:

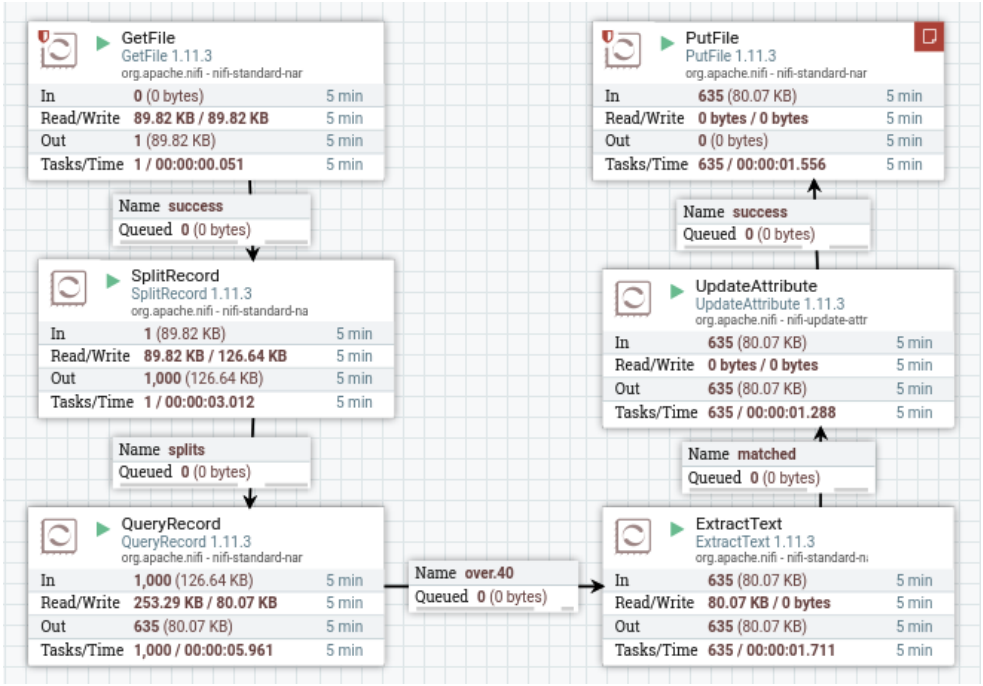

Figure 3.8 – The data pipeline you will build in this section

The following sections will walk you through building a data pipeline.

Reading a file with GetFile

The first step in your data pipeline is to read in the `data.csv` file. To do that, take the following steps:

1. Drag the **Processor** icon from the NiFi toolbar to the canvas. Search for **GetFile** and then select it.
2. To configure the `GetFile` processor, you must specify the input directory. In the Python examples earlier in this chapter, I wrote the `data.CSV` file to my home directory, which is `home/paulcrickard`, so this is what I will use for the input directory.
3. Next, you will need to specify a file filter. This field allows the NiFi expression language, so you could use **regular expressions** (**regex**) and specify any file ending with CSV – `[^\].*\.CSV` – but for this example, you can just set the value to `data.csv`.
4. Lastly, the **Keep Source File** property should be set to **true**. If you leave it as **false**, NiFi will delete the file once it has processed it. The complete configuration is shown in the following screenshot:

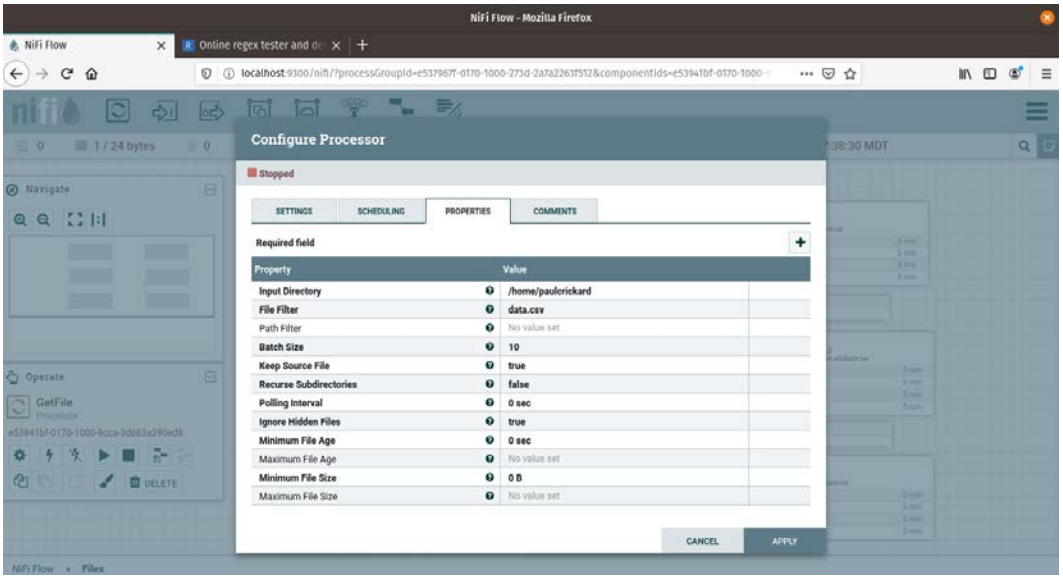

Figure 3.9 – GetFile processor configuration

Splitting records into distinct flowfiles

Now you can pass the success relationship from the `GetFile` processor to the `SplitRecord` processor:

1. The `SplitRecord` processor will allow you to separate each row into a separate flowfile. Drag and drop it on the canvas. You need to create a record reader and a record writer – NiFi already has several that you can configure. Click on the box next to **Record Reader** and select **Create new service**, as shown in the following screenshot:

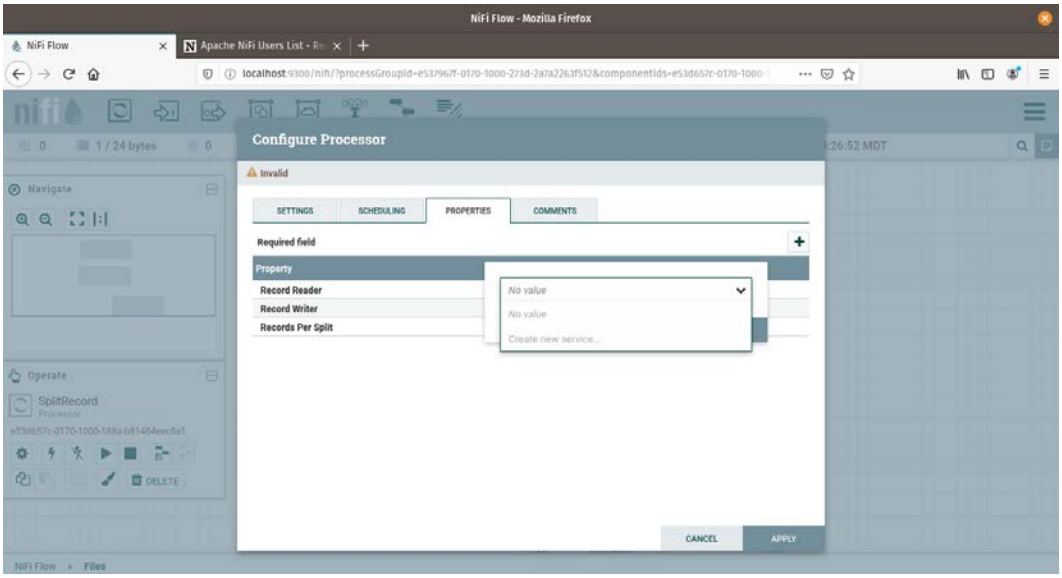

Figure 3.10 – A list of available readers

2. You will need to choose the type of reader. Select **CSVReader** from the dropdown. Select the dropdown for **Record Writer** and choose **CSVRecordSetWriter**:

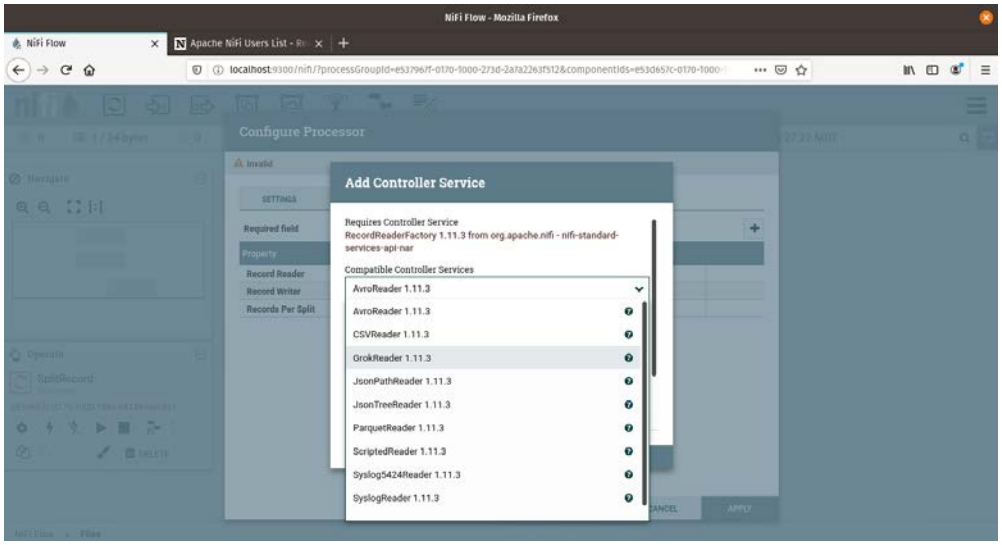

Figure 3.11 – A list of available readers

3. To configure **CSVReader** and **CSVRecordSetWriter**, click the arrow to the right of either one. This will open the **Files Configuration** window on the **CONTROLLER SERVICES** tab. You will see the screen shown in the following screenshot:

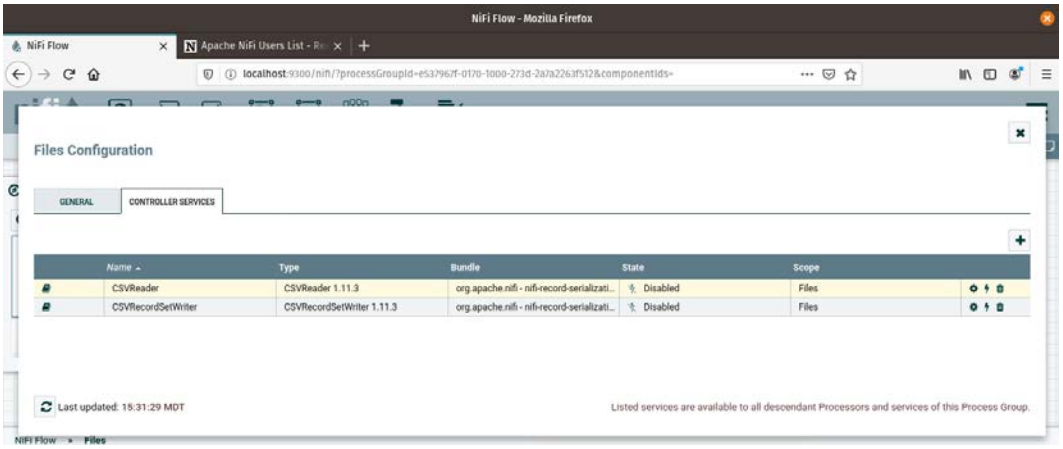

Figure 3.12 – Configuring the reader and writer

The three icons to the right are as follows:

- A gear for settings
- A lightning bolt for enabling and disabling the service (it is currently disabled)
- A trash can to delete it

Select the gear for **CSVReader**. The default configuration will work, except for the **Treat First Line as Header** property, which should be set to **true**. Click the gear for **CSVRecordSetWriter** and you can see the available properties. The defaults are sufficient in this example. Now, click the lightning bolt to enable the services.

Filtering records with the QueryRecord processor

You now have a pipeline that will read a CSV and split the rows into individual flowfiles. Now you can process each row with the `QueryRecord` processor. This processor will allow you to execute a SQL command against the flowfile. The contents of the new flowfile will be the results of the SQL query. In this example, you will select all records where the age of the person is over 40:

1. Drag and drop the `QueryRecord` processor to the canvas. To query the flowfile, you need to specify a record reader and writer. You have already created one of each of these and they are available in the dropdown now. The **Include Zero Record FlowFiles** property should be set to **false**. This property will route records that do not meet the criteria to the same relationship (which you do not want).
2. Lastly, click the plus sign in the right-hand corner and specify a property name in the popup. The name of the property will become a relationship when you create a connection from this processor. Name the property `over.40`. Then, the value popup will appear. This is where you will enter the SQL query. The results of the query will become the contents of the flowfile. Since you want the records of people over 40 years of age, the query is as follows:

   ```
   Select * from FlowFile where age > 40
   ```

   The `Select *` query is what returns the entire flowfile. If you only wanted the name of the person and for the field to be `full_name`, you could run the following SQL:

   ```
   Select name as full_name from FlowFile where age > 40
   ```

The point I am attempting to drive home here is that you can execute SQL and modify the flowfile to something other than the contents of the row – for example, running and aggregation and a group by.

Extracting data from a flowfile

The next processor will extract a value from the flowfile. That processer is `ExtractText`. The processor can be used on any flowfile containing text and uses regex to pull any data from the flowfile and assign it to an attribute.

To configure the processor, click the plus sign and name the property. You will extract the person name from the flowfile, so you can name the property name. The value will be regex and should be as follows:

```
\n([^,]*),
```

Without a full tutorial on regex, the preceding regex statement looks for a newline and a comma – `\n` and the comma at the end – and grabs the text inside. The parentheses say to take the text and return any characters that are not `^` or a comma. This regex returns the person's name. The flowfile contains a header of field names in CSV, a new line, followed by values in CSV. The `name` field is the first field on the second line – after the newline and before the first comma that specifies the end of the `name` field. This is why the regex looks for the text between the newline and the comma.

Modifying flowfile attributes

Now that you have pulled out the person name as an attribute, you can use the `UpdateAttribute` processor to change the value of existing attributes. By using this processor, you will modify the default filename attribute that NiFi has provided the flowfile all the way at the beginning in the `GetFile` processor. Every flowfile will have the filename `data.CSV`. If you try to write the flowfiles out to CSV, they will all have the same name and will either overwrite or fail.

Click the plus sign in the configuration for the `UpdateAttribute` processor and name the new property filename. The value will use the NiFi Expression Language. In the Expression Language, you can grab the value of an attribute using the format `${attribute name}`. So, to use the name attribute, set the value to `${name}`.

Saving a flowfile to disk

Using the `PutFile` processor, you can write the contents of a flowfile to disk. To configure the processor, you need to specify a directory in which to write the files. I will again use my home directory.

Next, you can specify a conflict resolution strategy. By default, it will be set to fail, but it allows you to overwrite an existing file. If you were running this data pipeline, aggregating data every hour and writing the results to files, maybe you would set the property to overwrite so that the file always holds the most current data. By default, the flowfile will write to a file on disk with the property filename as the filename.

Creating relationships between the processors

The last step is to make connections for specified relationships between the processors:

1. Grab the `GetFile` processor, drag the arrow to the `SplitRecord` processor, and check the relationship success in the popup.
2. From the `SplitRecord` processor, make a connection to the `QueryRecord` processor and select the relationship splits. This means that any record that was split will be sent to the next processor.
3. From `QueryRecord`, connect to the `ExtractText` processor. Notice the relationship you created is named `over.40`. If you added more SQL queries, you would get additional relationships. For this example, use the `over.40` relationship.
4. Connect `ExtractText` to the `UpdateAttribute` processor for the relationship matched.
5. Lastly, connect `UpdateAttribute` to the `PutFile` processor for the relationship success.

The data pipeline is now complete. You can click on each processor and select **Run** to start it – or click the run icon in the operate window to start them all at once.

When the pipeline is completed, you will have a directory with all the rows where the person was over 40. Of the 1,000 records, I have 635 CSVs named for each person. You will have different results based on what `Faker` used as the age value.

This section showed you how to read in a CSV file. You also learned how you can split the file into rows and then run queries against them, as well as how to modify attributes of a flowfile and use it in another processor. In the next section, you will build another data pipeline using JSON.

Working with JSON in NiFi

While having a different structure, working with JSON in NiFi is very similar to working with CSV. There are, however, a few processors for dealing exclusively with JSON. In this section, you will build a flow similar to the CSV example – read a file, split it into rows, and write each row to a file – but you will perform some more modifications of the data within the pipeline so that the rows you write to disk are different than what was in the original file. The following diagram shows the completed data pipeline:

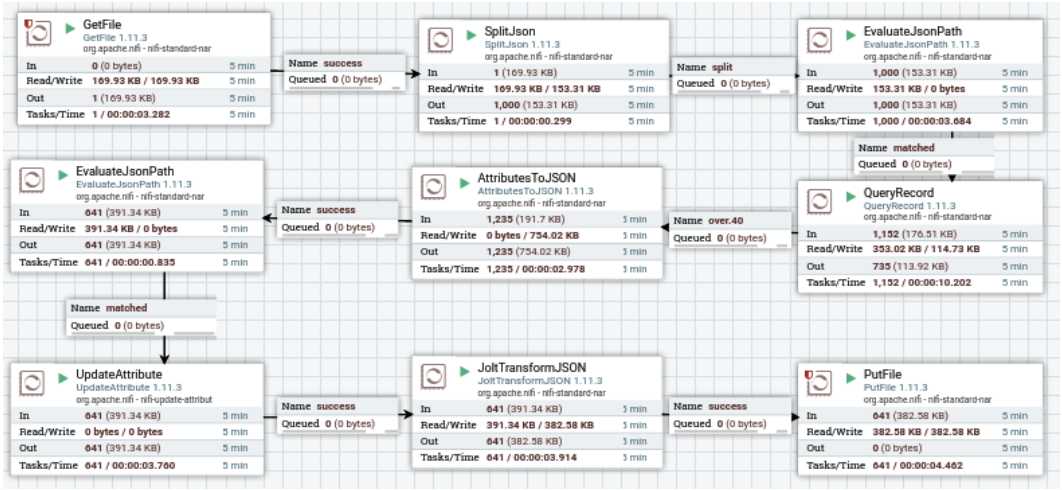

Figure 3.13 – The completed JSON data pipeline

To build the data pipeline, take the following steps:

1. Place the `GetFile` processor on to the canvas. To configure the processor, specify the **Input Directory** values as `home/paulcrickard` – and the **File Filter** value as `data.JSON`.
2. In the CSV example, you used the `SplitRecord` processor. Here, for JSON, you can use the `SplitJson` processor. You will need to configure the **JsonPath Expression** property. This property is looking for an array that contains JSON elements. The JSON file is in the following format:

```
{"records":[ { } ] }
```

Because each record is in an array, you can pass the following value to the **JsonPath Expression** property:

```
$.records
```

This will split records inside of the array, which is the result you want.

3. The records will now become individual flowfiles. You will pass the files to the `EvaluateJsonPath` processor. This processor allows you to extract values from the flowfile. You can either pass the results to the flowfile content or to an attribute. Set the value of the **Destination** property to `flowfile-attribute`. You can then select attributes to create using the plus sign. You will name the attribute, then specify the value. The value is the JSON path, and you use the format `$.key`. The configured processor is shown in the following screenshot:

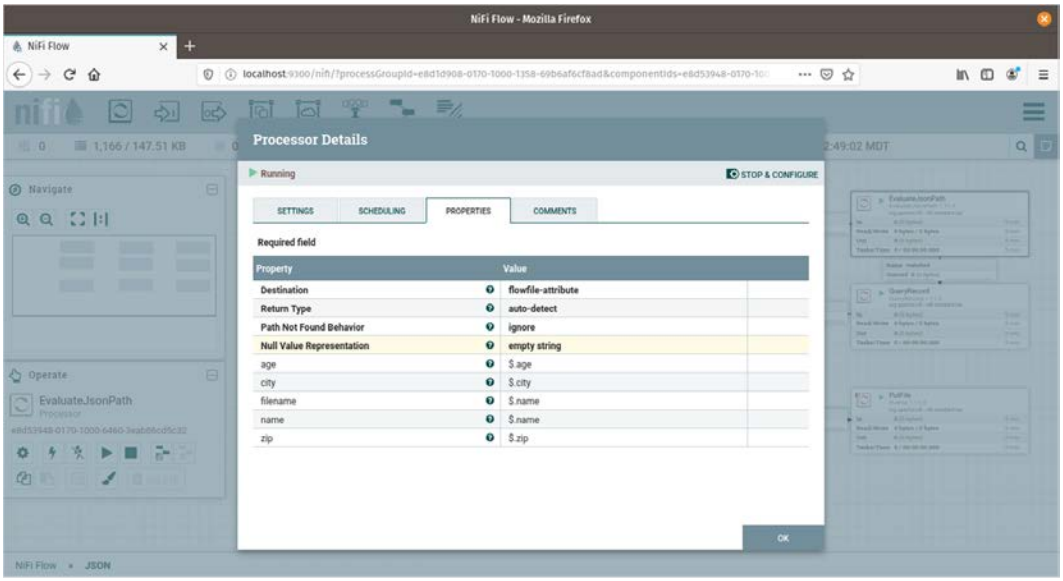

Figure 3.14 – Configuration for extracting values from the flowfile

These attributes will not be passed down the data pipeline with the flowfile.

4. Now, you can use the `QueryRecord` processor, just like you did with the CSV example. The difference with JSON is that you need to create a new record reader and recordset writer. Select the option to create a new service. Select **JsonTreeReader** and **JsonRecordsetWriter**. Click the arrow to go to the **Controller services** tab and click the lightning bolt to activate the services. The default configurations will work in this example. In the processor, add a new property using the plus sign. Name it `over.40` and set the value to the following:

```
Select * from FlowFile where age > 40
```

5. The next processor is the `AttributesToJSON` processor. This processor allows you to replace the flowfile content with the attributes you extracted in the `EvaluateJsonPath` processor shown in *step 3*. Set the **Destination** property to `flowfile-content`. This processor also allows you to specify a comma-separated list of attributes in the **Attributes List** property. This can come in handy if you only want certain attributes. In this example, you leave it blank and several attributes you do not extract will be added to the flowfile content. All of the metadata attributes that NiFi writes will now be a part of the flowfile. The flowfile will now look as in the following snippet:

```
### Run it at night ###
```

6. Using the `EvalueJsonPath` processor again, you will create an attribute named `uuid`. Now that the metadata from NiFi is in the flowfile, you have the unique ID of the flowfile. Make sure to set **Destination** to `flowfile-attribute`. You will extract it now so that you can pass it to the next processor – `UpdateAttribute`.
7. In the CSV example, you updated the filename using the `UpdateAttribute` processor. You will do the same here. Click on the plus sign and add an attribute named `filename`. Set the value to `${uuid}`.
8. One way to modify JSON using NiFi is through **Jolt transformations**. The **JSON Language for Transform** library allows you to modify JSON. A full tutorial on Jolt is beyond the scope of this book, but the processor allows you to select from several Jolt transformation DSLs. In this example, you will use a simple remove, which will delete a field. NiFi abbreviates the Jolt JSON because you have already specified what you are doing in the configuration. In the **Jolt Specification** property, enter the JSON, as shown in the following snippet:

```
{
    "zip": ""
}
```

   The preceding snippet will remove the `zip` field from the flowfile.

9. Lastly, use the `PutFile` processor to write each row to disk. Configure the **Directory** and **Conflict Resolution Strategy** properties. By setting the **Conflict Resolution Strategy** property to **ignore**, the processor will not warn you if it has already processed a file with the same name.

Create the connections and relationships between the processors:

- Connect `GetFile` to `SplitJson` for relationship success.
- Connect `SplitJson` to `EvaluateJsonPath` for relationship splits.
- Connect `EvaluateJsonPath` to `QueryRecord` for relationship matched.
- Connect `QueryRecord` to `AttributesToJSON` for relationship `over.40`.
- Connect `AttributesToJSON` to `UpdateAttribute` for relationship success.
- Connect `UpdateAttributes` to `JoltTransformJSON` for relationship success.
- Connect `JoltTransformJSON` to `PutFile` for relationship success.

Run the data pipeline by starting each processor or clicking **Run** in the operate box. When complete, you will have a subset of 1,000 files – all people over 40 – on disk and named by their unique ID.

Summary

In this chapter, you learned how to process CSV and JSON files using Python. Using this new skill, you have created a data pipeline in Apache Airflow by creating a Python function to process a CSV and transform it into JSON. You should now have a basic understanding of the Airflow GUI and how to run DAGs. You also learned how to build data pipelines in Apache NiFi using processors. The process for building more advanced data pipelines is the same, and you will learn the skills needed to accomplish this throughout the rest of this book.

In the next chapter, you will learn how to use Python, Airflow, and NiFi to read and write data to databases. You will learn how to use PostgreSQL and Elasticsearch. Using both will expose you to standard relational databases that can be queried using SQL and NoSQL databases that allow you to store documents and use their own query languages.

4
Working with Databases

In the previous chapter, you learned how to read and write text files. Reading log files or other text files from a data lake and moving them into a database or data warehouse is a common task for data engineers. In this chapter, you will use the skills you gained working with text files and learn how to move that data into a database. This chapter will also teach you how to extract data from relational and NoSQL databases. By the end of this chapter, you will have the skills needed to work with databases using Python, NiFi, and Airflow. It is more than likely that most of your data pipelines will end with a database and very likely that they will start with one as well. With these skills, you will be able to build data pipelines that can extract and load, as well as start and finish, with both relational and NoSQL databases.

In this chapter, we're going to cover the following main topics:

- Inserting and extracting relational data in Python
- Inserting and extracting NoSQL database data in Python
- Building database pipelines in Airflow
- Building database pipelines in NiFi

Inserting and extracting relational data in Python

When you hear the word **database**, you probably picture a relational database – that is, a database made up of tables containing columns and rows with relationships between the tables; for example, a purchase order system that has inventory, purchases, and customer information. Relational databases have been around for over 40 years and come from the relational data model developed by E. F. Codd in the late 1970s. There are several vendors of relational databases – including IBM, Oracle, and Microsoft – but all of these databases use a similar dialect of **SQL**, which stands for **Structured Query Language**. In this book, you will work with a popular open source database – **PostgreSQL**. In the next section, you will learn how to create a database and tables.

Creating a PostgreSQL database and tables

In *Chapter 2*, *Building Our Data Engineering Infrastructure*, you created a database in PostgreSQL using pgAdmin 4. The database was named `dataengineering` and you created a table named `users` with columns for name, street, city, ZIP, and ID. The database is shown in the following screenshot:

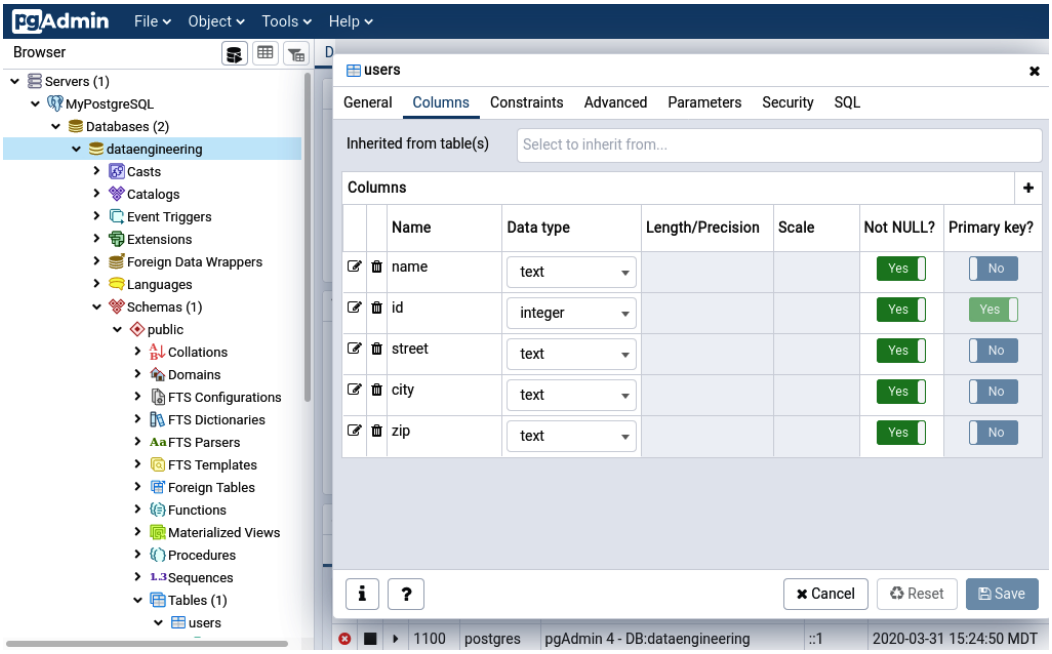

Figure 4.1 – The dataengineering database

If you have the database created, you can skip this section, but if you do not, this section will quickly walk you through creating one.

To create a database in PostgreSQL with pgAdmin 4, take the following steps:

1. Browse to `http://localhost/pgadmin4` and log in using the account you created during the installation of `pgAdmin` in *Chapter 2*, *Building Our Data Engineering Infrastructure*.
2. Expand the server icon in the **Browser** pane. Right-click on the **MyPostgreSQL** icon and select **Create | Database**.
3. Name the database `dataengineering`. You can leave the user as `postgres`.
4. Expand the `dataengineering` icon, then expand **Schemas**, then **public**, then **Tables**. Right-click on **Tables**, then click **Create | Table**.
5. Name the table `users`. Click the **Columns** tab and then, using the plus sign on the right, create columns to match the preceding screenshot of the database. The columns' names and types will be as follows:

   a) `name`: `text`

   b) `id`: `integer`

   c) `street`: `text`

   d) `city`: `text`

   e) `zip`: `text`

Now you have a database and a table created in PostgreSQL and can load data using Python. You will populate the table in the next section.

Inserting data into PostgreSQL

There are several libraries and ways to connect to a database in Python – `pyodbc`, `sqlalchemy`, `psycopg2`, and using an API and requests. In this book, we will use the `psycopg2` library to connect to PostgreSQL because it is built specifically to connect to PostgreSQL. As your skills progress, you may want to look into tools such as **SQLAlchemy**. SQLAlchemy is a toolkit and an object-relational mapper for Python. It allows you to perform queries in a more Pythonic way – without SQL – and to map Python classes to database tables.

Installing psycopg2

You can check whether you have `psycopg2` installed by running the following command:

```
python3 -c "import psycopg2; print(psycopg2.__version__)"
```

The preceding command runs `python3` with the command flag. The flag tells Python to run the commands as a Python program. The quoted text imports `psycopg2` and then prints the version. If you receive an error, it is not installed. You should see a version such as 2.8.4 followed by some text in parentheses. The library should have been installed during the installation of Apache Airflow because you used all the additional libraries in *Chapter 2*, *Building Our Data Engineering Infrastructure*.

If it is not installed, you can add it with the following command:

```
pip3 install psycopg2
```

Using `pip` requires that there are additional dependencies present for it to work. If you run into problems, you can also install a precompiled binary version using the following command:

```
pip3 install psycopg2-binary
```

One of these two methods will get the library installed and ready for us to start the next section.

Connecting to PostgreSQL with Python

To connect to your database using `psycopg2`, you will need to create a connection, create a cursor, execute a command, and get the results. You will take these same steps whether you are querying or inserting data. Let's walk through the steps as follows:

1. Import the library and reference it as `db`:

   ```
   import psycopg2 as db
   ```

2. Create a connection string that contains the host, database, username, and password:

   ```
   conn_string="dbname='dataengineering' host='localhost'
   user='postgres' password='postgres'"
   ```

3. Create the connection object by passing the connection string to the `connect()` method:

```
conn=db.connect(conn_string)
```

4. Next, create the cursor from the connection:

```
cur=conn.cursor()
```

You are now connected to the database. From here, you can issue any SQL commands. In the next section, you will learn how to insert data into PostgreSQL

Inserting data

Now that you have a connection open, you can insert data using SQL. To insert a single person, you need to format a SQL `insert` statement, as shown:

```
query = "insert into users (id,name,street,city,zip)
values({},'{}','{}','{}','{}')".format(1,'Big Bird','Sesame
Street','Fakeville','12345')
```

To see what this query will look like, you can use the `mogrify()` method.

> **What is mogrify?**
>
> According to the `psycopg2` docs, the `mogrify` method will return a query string after arguments binding. The string returned is exactly the one that would be sent to the database running the `execute()` method or similar. In short, it returns the formatted query. This is helpful as you can see what you are sending to the database, because your SQL query can often be a source of errors.

Pass your query to the `mogrify` method:

```
cur.mogrify(query)
```

The preceding code will create a proper SQL `insert` statement; however, as you progress, you will add multiple records in a single statement. To do so, you will create a tuple of tuples. To create the same SQL statement, you can use the following code:

```
query2 = "insert into users (id,name,street,city,zip)
values(%s,%s,%s,%s,%s)"
data=(1,'Big Bird','Sesame Street','Fakeville','12345')
```

Notice that in `query2`, you did not need to add quotes around strings that would be passed in as you did in `query` when you used { }. Using the preceding formatting, `psycopg2` will handle the mapping of types in the query string. To see what the query will look like when you execute it, you can use `mogrify` and pass the data along with the query:

```
cur.mogrify(query2,data)
```

The results of `mogrify` on `query` and `query2` should be identical. Now, you can execute the query to add it to the database:

```
cur.execute(query2,data)
```

If you go back to pgAdmin 4, right-click on the **users** table, then select **View/Edit Data | All Rows**, you can see that no data has been added to the table. Why is that? Did the code fail? It did not. When you execute any statement that modifies the database, such as an `insert` statement, you need to make it permanent by committing the transaction using the following code:

```
conn.commit()
```

Now, in pgAdmin 4, you should be able to see the record, as shown in the following screenshot:

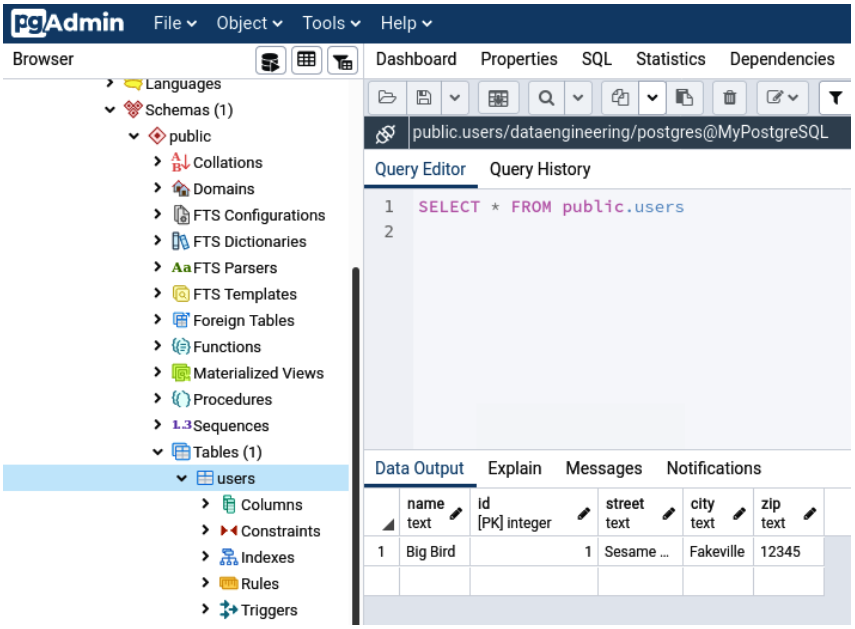

Figure 4.2 – Record added to the database

The record is now added to the database and visible in pgAdmin 4. Now that you have entered a single record, the next section will show you how to enter multiple records.

Inserting multiple records

To insert multiple records, you could loop through data and use the same code shown in the preceding section, but this would require a transaction per record in the database. A better way would be to use a single transaction and send all the data, letting `psycopg2` handle the bulk insert. You can accomplish this by using the `executemany` method. The following code will use `Faker` to create the records and then `executemany()` to insert them:

1. Import the needed libraries:

```
import psycopg2 as db
from faker import Faker
```

2. Create the `faker` object and an array to hold all the data. You will initialize a variable, `i`, to hold an ID:

```
fake=Faker()
data=[]
i=2
```

3. Now, you can look, iterate, and append a fake tuple to the array you created in the previous step. Increment `i` for the next record. Remember that in the previous section, you created a record for `Big Bird` with an ID of `1`. That is why you will start with `2` in this example. We cannot have the same primary key in the database table:

```
for r in range(1000):
    data.append((i,fake.name(),fake.street_address(),
                 fake.city(),fake.zipcode()))
    i+=1
```

4. Convert the array into a tuple of tuples:

```
data_for_db=tuple(data)
```

5. Now, you are back to the `psycopg` code, which will be similar to the example from the previous section:

```
conn_string="dbname='dataengineering' host='localhost'
user='postgres' password='postgres'"
conn=db.connect(conn_string)
cur=conn.cursor()
query = "insert into users (id,name,street,city,zip)
values(%s,%s,%s,%s,%s)"
```

6. You can print out what the code will send to the database using a single record from the `data_for_db` variable:

```
print(cur.mogrify(query,data_for_db[1]))
```

7. Lastly, use `executemany()` instead of `execute()` to let the library handle the multiple inserts. Then, commit the transaction:

```
cur.executemany(query,data_for_db)
conn.commit()
```

Now, you can look at pgAdmin 4 and see the 1,000 records. You will have data similar to what is shown in the following screenshot:

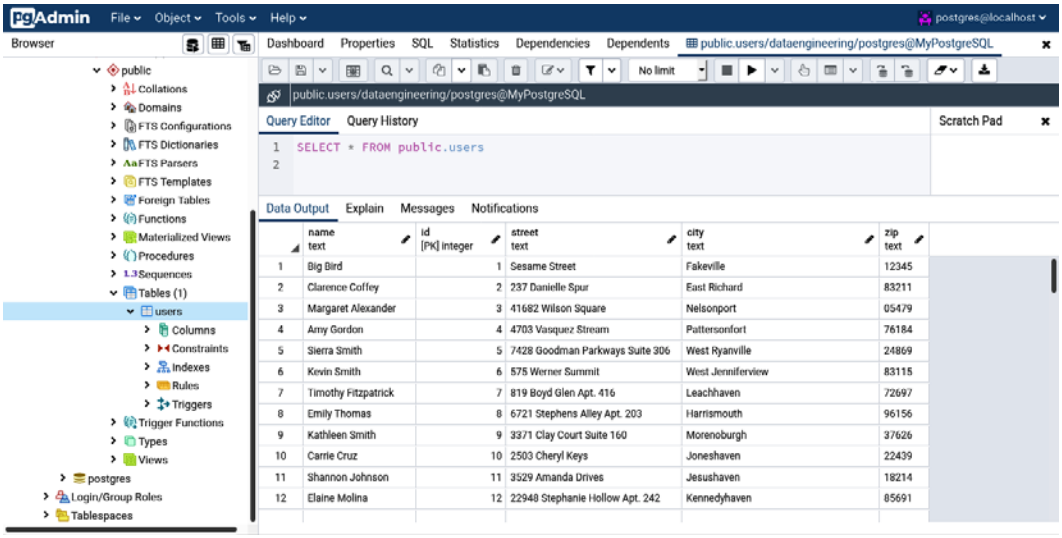

Figure 4.3 – 1,000 records added to the database

Your table should now have 1,001 records. Now that you can insert data into PostgreSQL, the next section will show you how to query it in Python.

Extracting data from PostgreSQL

Extracting data using `psycopgs` follows the exact same procedure as inserting, the only difference being that you will use a `select` statement instead of `insert`. The following steps show you how to extract data:

1. Import the library, then set up your connection and cursor:

```
import psycopg2 as db
conn_string="dbname='dataengineering' host='localhost'
user='postgres' password='postgres'"
conn=db.connect(conn_string)
cur=conn.cursor()
```

2. Now, you can execute a query. In this example, you will select all records from the `users` table:

```
query = "select * from users"
cur.execute(query)
```

3. Now, you have an iterable object with the results. You can iterate over the cursor, as shown:

```
for record in cur:
    print(record)
```

4. Alternatively, you could use one of the `fetch` methods:

```
cur.fetchall()
cur.fetchmany(howmany)  # where howmany equals the number
of records you want returned
cur.fetchone()
```

5. To grab a single record, you can assign it to a variable and look at it. Note that even when you select one record, the cursor returns an array:

```
data=cur.fetchone()
print(data[0])
```

To start the data pipeline, you will use the `GenerateFlowFile` processor. Drag and drop the processor on the canvas. Double-click on it to change the configuration. In the **Settings** tab, name the processor. I have named it `Start Flow Fake Data`. This lets us know that this processor sends fake data just to start the flow. The configuration will use all the defaults and look like the following screenshot:

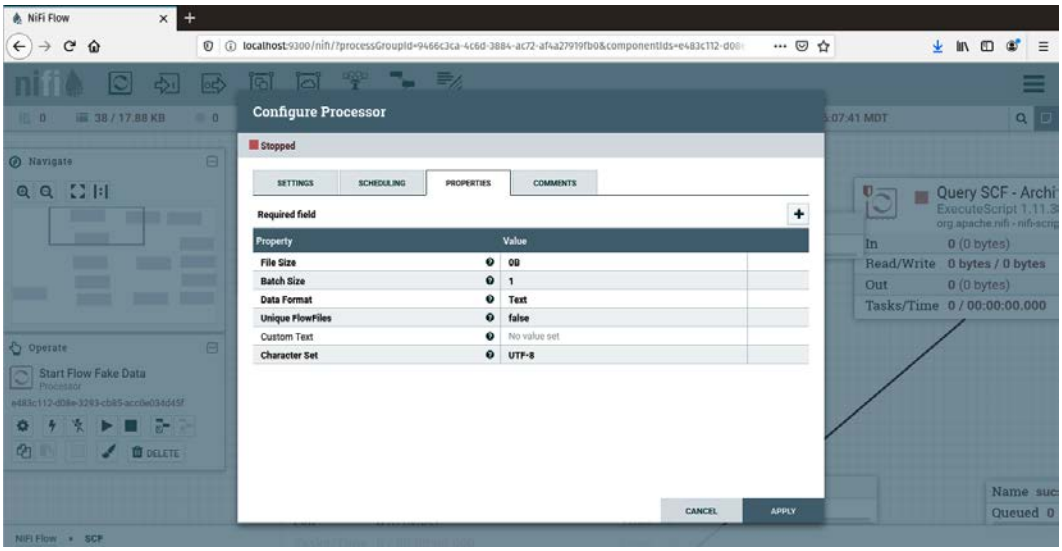

Figure 6.3 – Configuring the GenerateFlowfile processor

Lastly, in the **SCHEDULING** tab, set the processor to run at your desired interval. I use 8 because I do not want to overwhelm the API.

The processor, when running, will generate a single flowfile with 0 bytes of data. It is empty, but it does contain metadata generated by NiFi. However, this empty flowfile will do the trick and start the next processor. That is where the work begins.

Querying SeeClickFix

In the previous NiFi examples, you did not use any code, just configurations to make the processor do what you needed. We could do that in this pipeline. However, now is a good time to introduce coding using Python – Jython – into your pipelines.

Drag and drop the `ExecuteScript` processor to the canvas. Double-click on it to edit the configuration. Starting with the **Settings** tab, name it something that makes sense to you – I named it `Query SCF` so that I know it queries **SeeClickFix**. In the **Properties** tab, set **Script Engine** to **Python**. In the **Script Body** parameter, you will write the Python code that the processor will execute. The query steps are as follows:

1. You need to import the required libraries. The following code is the libraries that you will always need to include:

```
import java.io
from org.apache.commons.io import IOUtils
from java.nio.charset import StandardCharsets
from org.apache.nifi.processor.io import StreamCallback
from org.python.core.util import StringUtil
```

2. Next, you will create the class that will be called to handle the work. The `process` function will contain the code that will perform the task:

```
class ModJSON(StreamCallback):
  def __init__(self):
        pass
  def process(self, inputStream, outputStream):
    # Task Goes Here
```

3. Lastly, assume that no errors have occurred, and check whether there is a flowfile. If there is one, write the flowfile calling the class. Next, check whether an error occurred. If there was an error, you will send the flowfile to the failure relationship, otherwise, send it to the success relationship:

```
errorOccurred=False
flowFile = session.get()
if (flowFile != None):
  flowFile = session.write(flowFile, ModJSON())
  #flowFile = session.putAttribute(flowFile)
  if(errorOccurred):
    session.transfer(flowFile, REL_FAILURE)
 else:
    session.transfer(flowFile, REL_SUCCESS)
```

The preceding code is the boiler plate for any Python `ExecuteScript` processor. The only thing you will need to change will be in the process function, which we will do in the steps that follow.

Because NiFi uses Jython, you can add many Python libraries to the Jython environment, but that is beyond the scope of this book. For now, you will use the standard libraries.

4. To make a call to the SeeClickFix API, you will need to import the `urllib` libraries and `json`, as shown:

```
import urllib
import urllib2
import json
```

5. Next, you will put the code in the `process` function. The code will be a `try except` block that makes a request to the HTTP endpoint and writes out the response to `outputStream`. If there was an error, the `except` block will set `errorOccurred` to `True` and this will trigger the rest of the code to send the flowfile to the `Failure` relationship. The only line in the `try` block that is not standard Python for using `urllib` is `outputStream.write()`. This is where you write to the flowfile:

```
    try:
        param = {'place_url':'bernalillo-county',
                 'per_page':'100'}
        url = 'https://seeclickfix.com/api/v2/issues?' +
               urllib.urlencode(param)
        rawreply = urllib2.urlopen(url).read()
        reply = json.loads(rawreply)

        outputStream.write(bytearray(json.dumps(reply,
               indent=4).encode('utf-8')))
    except:
        global errorOccurred
        errorOccurred=True

        outputStream.write(bytearray(json.dumps(reply,
               indent=4).encode('utf-8')))
```

The preceding code, when successful, will output a JSON flowfile. The contents of the flowfile will contain some metadata and an array of issues. The two pieces of metadata we will be interested in are **page** and **pages**.

You have grabbed the first 100 issues for Bernalillo County, and will pass this flowfile to two different processors – `GetEveryPage` and `SplitJson`. We will follow the `SplitJson` path, as this path will send the data to Elasticsearch.

Transforming the data for Elasticsearch

The following are the steps for transforming data for Elasticsearch:

1. Drag and drop the `SplitJson` processor to the canvas. Double-click on it to modify the properties. In the **Properties** tab, set the **JsonPath Expression** property to **$.issues**. This processor will now split the 100 issues into their own flowfiles.
2. Next, you need to add coordinates in the format expected by NiFi. We will use an `x`, `y` string named `coords`. To do that, drag and drop an `ExecuteScript` processor to the canvas. Double-click on it and click the **Properties** tab. Set the **Script Engine** to **Python**. The **Script Body** property will contain the standard boiler plate, plus the `import json` statement.
3. The `process` function will convert the input stream to a string. The input stream is the flowfile contents from the previous processor. In this case, it is a single issue. Then it will use the `json` library to load it as `json`. You then add a field named `coords` and assign it the value of a concatenated string of the `lat` and `lng` fields in the flowfile JSON. Lastly, you write the JSON back to the output stream as a new flowfile:

```
  def process(self, inputStream, outputStream):
      try:
          text = IOUtils.toString(inputStream,
                                  StandardCharsets.UTF_8)
          reply=json.loads(text)

  reply['coords']=str(reply['lat'])+','+str(reply['lng'])
          d=reply['created_at'].split('T')
          reply['opendate']=d[0]
          outputStream.write(bytearray(json.dumps(reply,
                               indent=4).encode('utf-8')))
      except:
```

```
            global errorOccurred
            errorOccurred=True
            outputStream.write(bytearray(json.dumps(reply,
                                    indent=4).encode('utf-8')))
```

Now you have a single issue, with a new field called `coords`, that is a string format that Elasticsearch recognizes as a geopoint. You are almost ready to load the data in Elasticsearch, but first you need a unique identifier.

4. To create the equivalent of a primary key in Elasticsearch, you can specify an ID. The JSON has an ID for each issue that you can use. To do so, drag and drop the `EvaluateJsonPath` processor on to the canvas. Double-click on it and select the **Properties** tab. Clicking the plus sign in the upper-right corner, add a property named `id` with the value of `$.id`. Remember that `$.` allows you to specify a JSON field to extract. The flowfile now contains a unique ID extracted from the JSON.
5. Drag and drop the `PutElasticsearchHttp` processor on to the canvas. Double-click on it to edit the properties. Set the **Elasticsearch URL** property to `http://localhost:9200`. In the optional **Identifier Attribute** property, set the value to **id**. This is the attribute you just extracted in the previous processor. Set the **Index** to **SCF** (short for **SeeClickFix**), and the **Type** to **doc**. Lastly, you will set the **Index Operation** property to **upsert**. In Elasticsearch, **upsert** will index the document if the ID does not already exist, and it will update if the ID exists, and the data is different. Otherwise, nothing will happen, and the record will be ignored, which is what you want if the data is already the same.

The issues are now being loaded in Elasticsearch, and if you were to check, you will have 100 documents in your `scf` index. But there are a lot more than 100 records in the SeeClickFix data for Bernalillo County; there are 44 pages of records (4,336 issues) according to the metadata from the `QuerySCF` processor.

The following section will show you how to grab all the data.

Getting every page

When you queried SeeClickFix, you sent the results to two paths. We took the `SplitJson` path. The reason for this is because on the initial query, you got back 100 issues and how many pages of issues exist (as part of the metadata). You sent the issues to the `SplitJson` path, because they were ready to process, but now you need to do something with the number of pages. We will do that by following the `GetEveryPage` path.

Drag and drop an `ExecuteScript` processor on to the canvas. Double-click on it to edit the **Properties** tab. Set the **Script Engine** property to **Python** and the **Script Body** will include the standard boiler plate – including the imports for the `urllib` and `json` libraries.

The `process` function will convert the input stream to JSON, and then it will load it using the `json` library. The main logic of the function states that if the current page is less than or equal to the total number of pages, call the API and request the next page (`next_page_url`), and then write out the JSON as a flowfile. Otherwise, it stops. The code is as follows:

```
try:
        text = IOUtils.toString(inputStream,
                                StandardCharsets.UTF_8)
        asjson=json.loads(text)
        if asjson['metadata']['pagination']
        ['page']<=asjson['metadata']['pagination']['pages']:
          url = asjson['metadata']['pagination']
                                  ['next_page_url']
          rawreply = urllib2.urlopen(url).read()
          reply = json.loads(rawreply)
          outputStream.write(bytearray(json.dumps(reply,
                              indent=4).encode('utf-8')))
        else:
          global errorOccurred
          errorOccurred=True
          outputStream.write(bytearray(json.dumps(asjson,
                              indent=4).encode('utf-8')))
    except:
        global errorOccurred
        errorOccurred=True
        outputStream.write(bytearray(json.dumps(asjson,
                            indent=4).encode('utf-8')))
```

You will connect the relationship success for this processor to the `SplitJson` processor in the last path we took. The flowfile will be split on issues, coordinates added, the ID extracted, and the issue sent to Elasticsearch. However, we need to do this 42 times.

To keep processing pages, you need to connect the success relationship to itself. That's right; you can connect a processor to itself. When you processed the first page through this processor, the next page was 2. The issues were sent to `SplitJson`, and back to this processor, which said the current page is less than 44 and the next page is 3.

You now have an Elasticsearch index with all of the current issues from SeeClickFix. However, the number of issues for Bernalillo County is much larger than the set of current issues – there is an archive. And now that you have a pipeline pulling new issues every 8 hours, you will always be up to date, but you can backfill Elasticsearch with all of the archived issues as well. Then you will have the full history of issues.

Backfilling data

To backfill the `SCF` index with historic data only requires the addition of a single parameter to the `params` object in the `QuerySCF` processor. To do that, right-click on the `QuerySCF` processor and select **copy**. Right-click on a blank spot of canvas, and then select **paste**. Double-click the copied processor and, in the **Settings** tab, rename it as `QuerySCFArchive`. In the **Properties** tab, modify the **Script Body** parameter, changing the `params` object to the following code:

```
param = {'place_url':'bernalillo-county', 'per_page': '100',
'status':'Archived'}
```

The `status` parameter was added with the value `Archived`. Now, connect the `GenerateFlowfile` processor to this backfill processor to start it. Then, connect the processor to the `SplitJson` processor for the success relationship. This will send the issues to Elasticsearch. But you need to loop through all the pages, so connect the processor to the `GetEveryPage` processor too. This will loop through the archives and send all the issues to Elasticsearch. Once this pipeline finishes, you can stop the `QuerySCFArchive` processor.

When you have a system that is constantly adding new records – like a transactional system – you will follow this pattern often. You will build a data pipeline to extract the recent records and extract the new records at a set interval – daily or hourly depending on how often the system updates or how much in real time you need it to be. Once your pipeline is working, you will add a series of processors to grab all the historic data and backfill your warehouse. You may not need to go back to the beginning of time, but in this case, there were sufficiently few records to make it feasible.

You will also follow this pattern if something goes wrong or if you need to populate a new warehouse. If your warehouse becomes corrupted or you bring a new warehouse online, you can rerun this backfill pipeline to bring in all the data again, making the new database complete. But it will only contain current state. The next chapter deals with production pipelines and will help you solve this problem by improving your pipelines. For now, let's visualize your new Elasticsearch index in Kibana.

Building a Kibana dashboard

Now that your SeeClickFix data pipeline has loaded data in Elasticsearch, it would be nice to see the results of the data, as would an analyst. Using Kibana, you can do just that. In this section, you will build a Kibana dashboard for your data pipeline.

To open Kibana, browse to `http://localhost:5601` and you will see the main window. At the bottom of the toolbar (on the left of the screen; you may need to expand it), click the management icon at the bottom. You need to select **Create new Index Pattern** and enter `scf*`, as shown in the following screenshot:

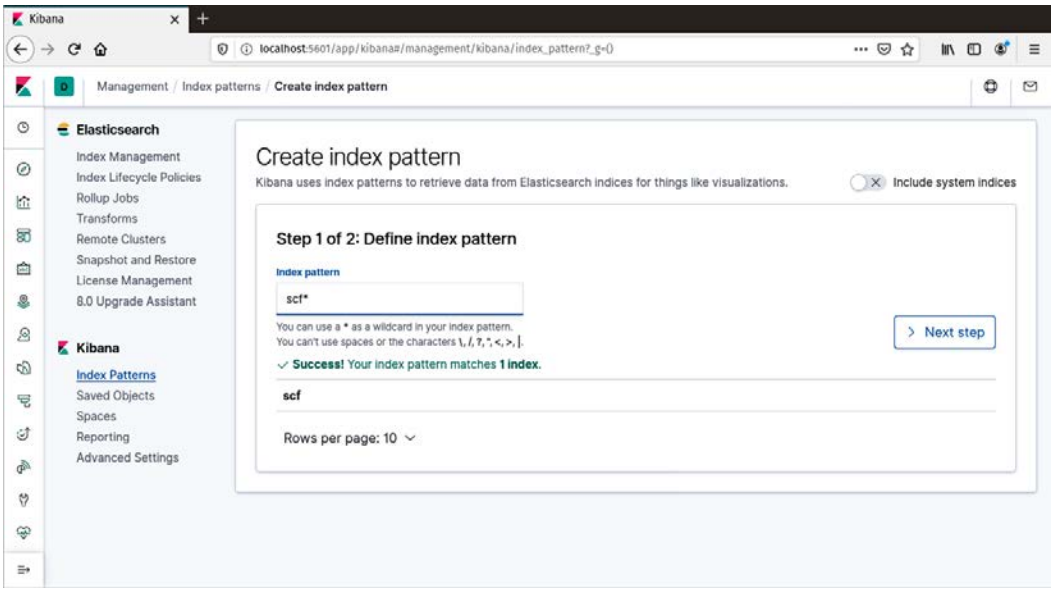

Figure 6.4 – Creating the index pattern in Kibana

When you click the next step, you will be asked to select a **Time Filter field name**. Because there are several fields with times in them, and they are in a format that is already recognizable by Elasticsearch, they will be indexed as such, and you can select a primary time filter. The field selected will be the default field used in screens such as **Discovery** when a bar chart preview of the data is displayed by time, and when you use a time filter in visualizations or dashboards. I have selected `created_at`, as shown in the following screenshot:

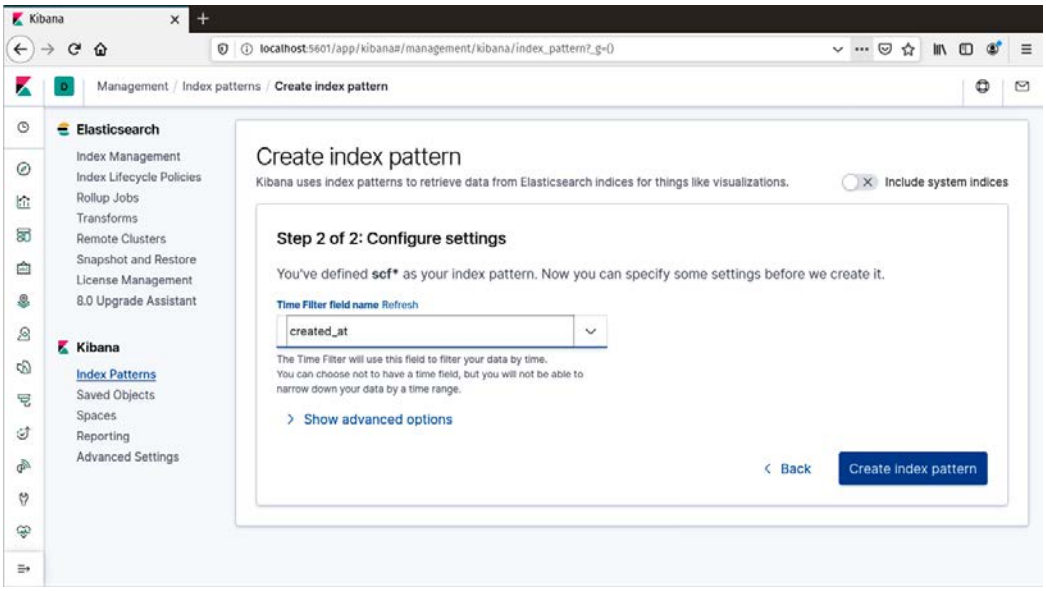

Figure 6.5 – Selecting the Time Filter field

Once you have created the index in Kibana, you can move on to visualizations.

Creating visualizations

To create visualizations, select the visualization icon in the toolbar. Select **Create Visualization** and you will see a variety of types available, as shown in the following screenshot:

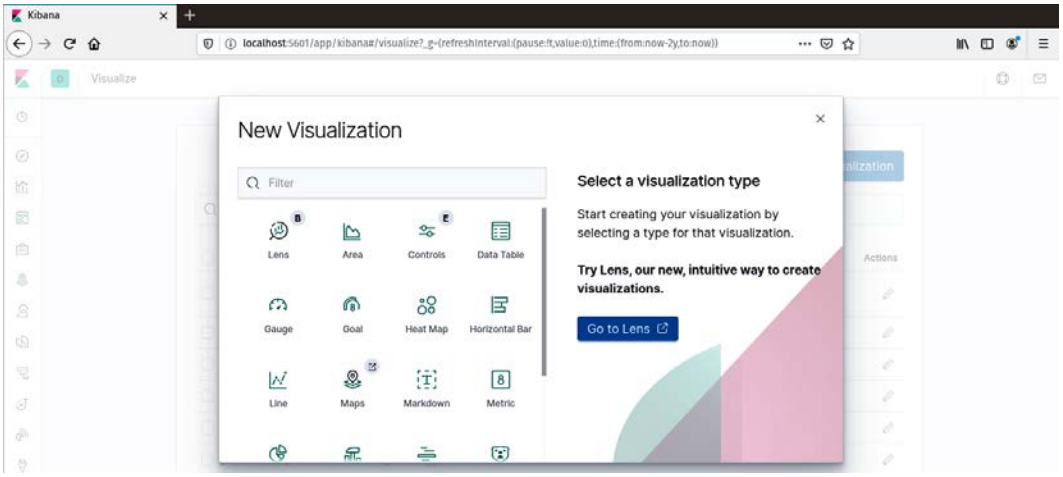

Figure 6.6 – Available visualization types

You will see the **Lens** type, which is a **Beta** visualization, as well as **Controls** and **Vega**, which are **Experimentals**. For now, select the **Vertical Bar** chart. When asked for a source, choose `scf` — this will apply to all visualizations in this chapter. Leave the y axis as **Count**, but add a new bucket and select the x axis. For **Aggregations**, choose **Date Histogram**. The field is `created_at` and the interval will be **Monthly**. You will see a chart as shown in the following screenshot (yours may vary):

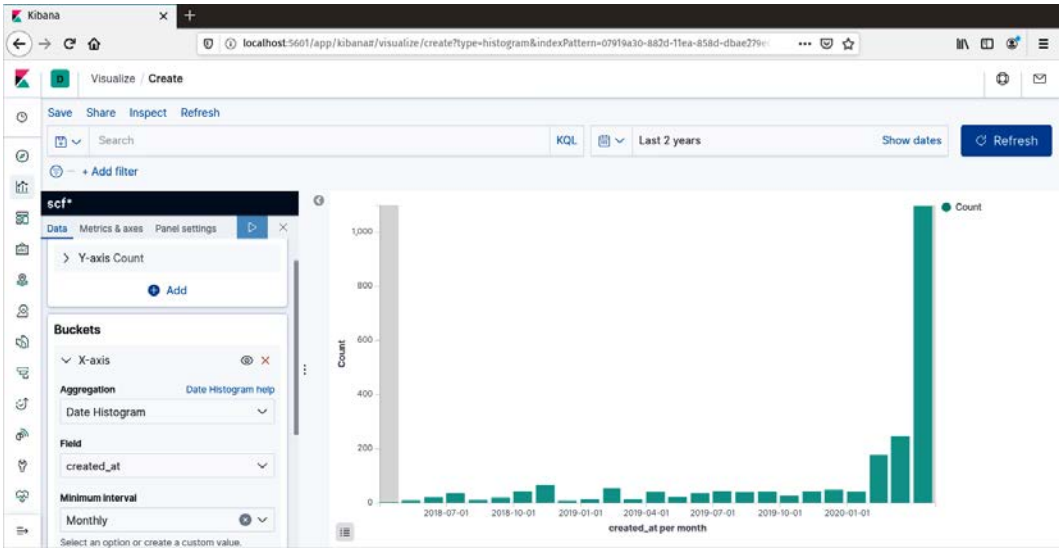

Figure 6.7 – Bar chart of created_at counts by month

Save the bar chart and name it `scf-bar`, or anything that you will be able to associate with the SeeClickFix data.

Next, select visualization again and choose metric. You will only add a custom label under the **Metrics** options. I chose **Issues**. By doing this, you remove the default count that gets placed under the numbers in the metric. This visualization is giving us a count of issues and will change when we apply filters in the dashboard. The configuration is shown in the following screenshot:

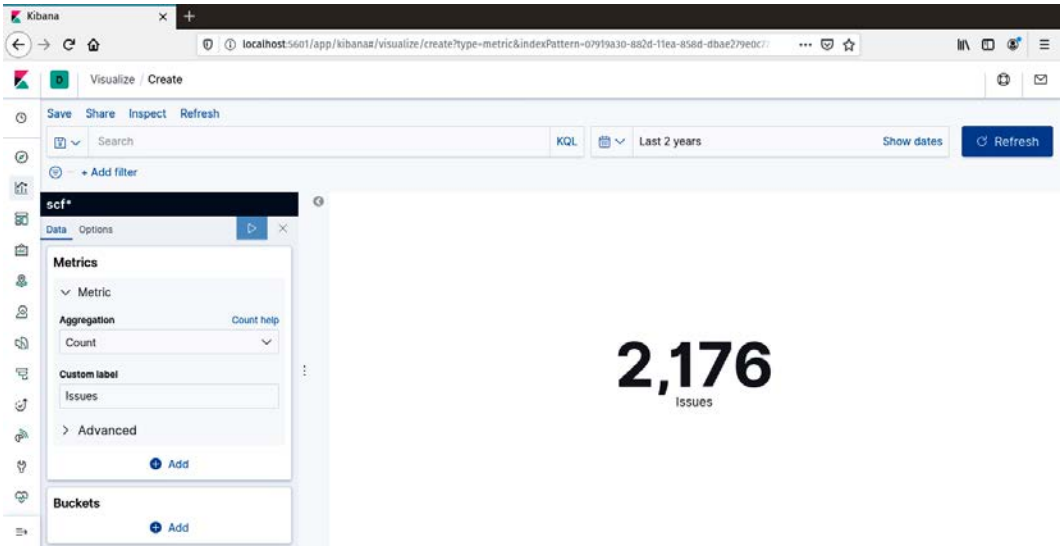

Figure 6.8 – Metrics visualization configuration

Again, save the visualization using any convention, or prefix it with `scf`, as I have done.

For the next visualization, select a pie chart – which will default to a donut. Under **Buckets**, select **Split slices**. For **Aggregations**, select **Terms**. And for **Field**, select **request_type.title.keyword**. Leave the rest of the defaults set. This will give you the top five titles. The results are shown in the following screenshot:

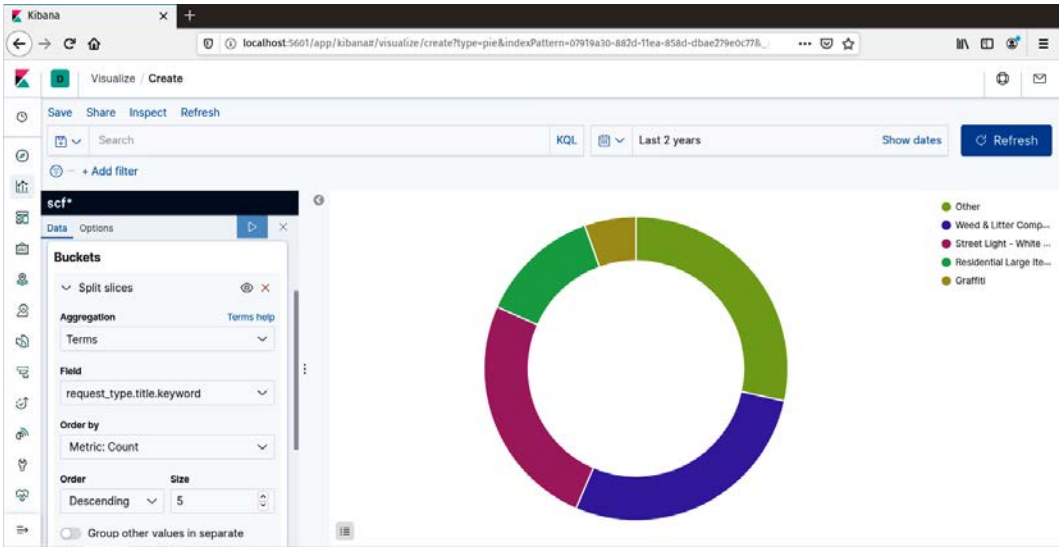

Figure 6.9 – Top five issue titles

While not a visualization, **Markdown** can add value to your dashboard by providing some context or a description. Select **Markdown** from the visualization options. You can enter Markdown in the left pane and, by clicking the run symbol, see the preview in the right pane. I have just added an H1, some text, and a bullet list, as shown in the following screenshot:

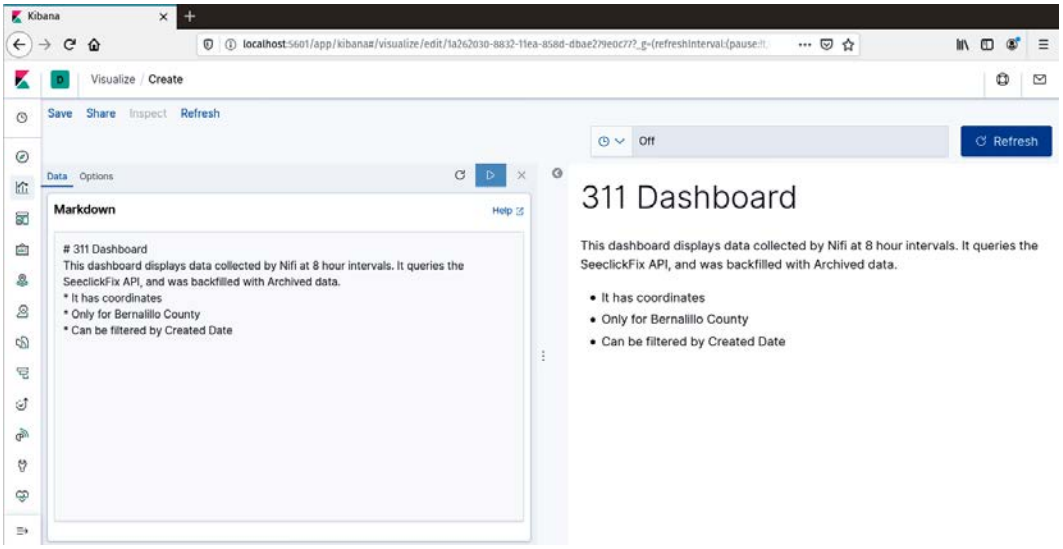

Figure 6.10 – Using the Markdown editor

The last visualization, **Map**, has an arrow because maps have their own place on the toolbar, and you can do a lot more with them than the other visualizations. For now, you can select **Map** from either location. You will select **Create Map**, and when prompted for the index pattern, select `scf`. Once on the map screen, select **Add Layer** and the source will be **Documents**. This allows you to select an index. The following screenshot shows what you should see:

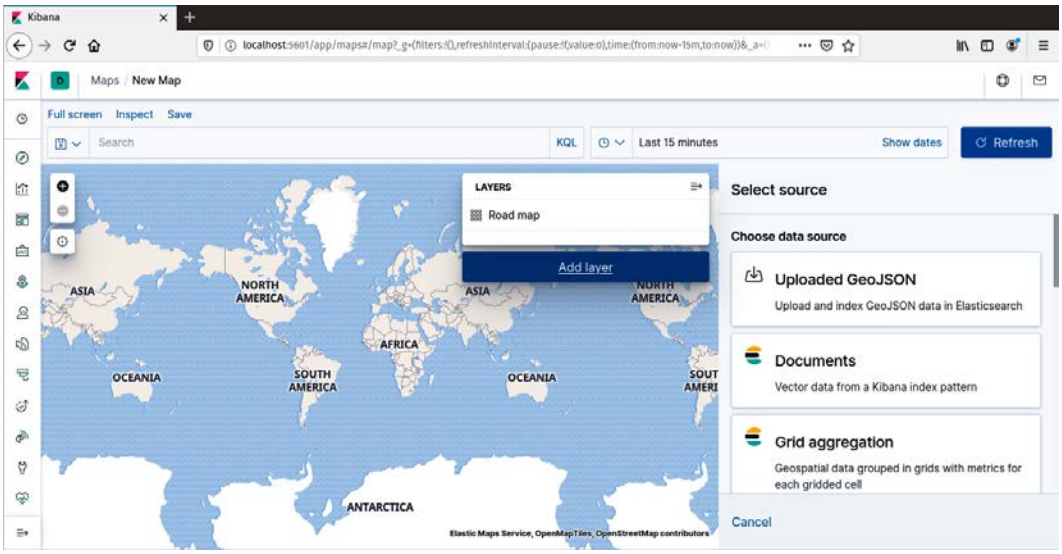

Figure 6.11 – Adding a new layer with the source being documents

When you select `scf` as the index pattern, Kibana will recognize the appropriate field and add the data to the map. Your map will be blank, and you may wonder went wrong. Kibana sets the time filter to the last 15 minutes, and you do not have data newer than the last 8 hours. Set the filter to a longer time frame, and the data will appear if the `create_at` field is in the window. The results are shown in the following screenshot:

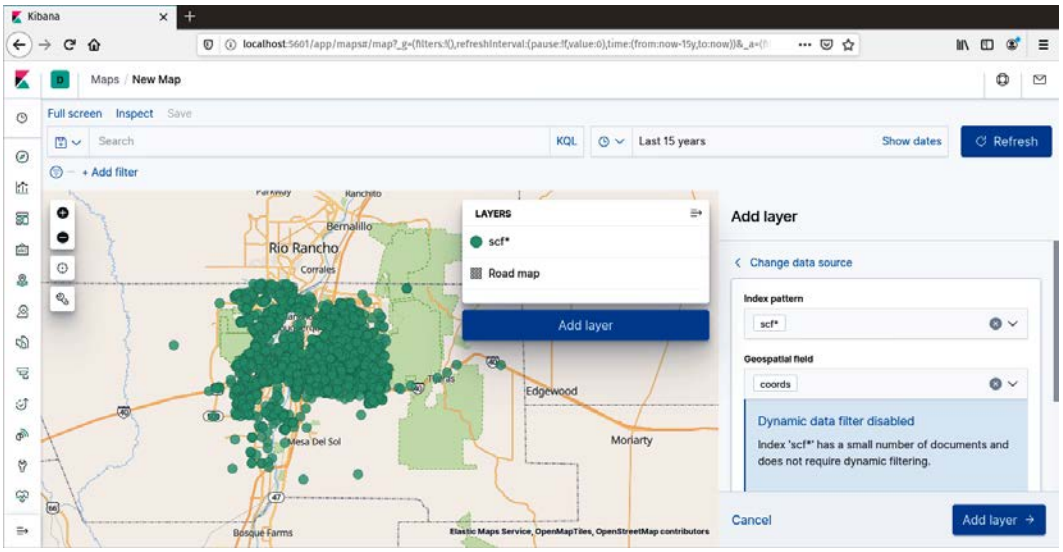

Figure 6.12 – A map visualization from an Elasticsearch index

Now that you have created visualizations from your data, you can now move on to combining them into a dashboard. The next section will show you how.

Creating a dashboard

To build a dashboard, select the dashboard icon on the toolbar. You will then select **Create a new dashboard** and add visualizations. If this is the first dashboard, you may see text asking whether you want to add an existing item. Add an item and then, in the search bar, type `scf` – or any of the names you used to save your visualizations. Adding them to the dashboard, you can then position them and resize them. Make sure to save your dashboard once it is set up. I have built the dashboard shown in the following screenshot:

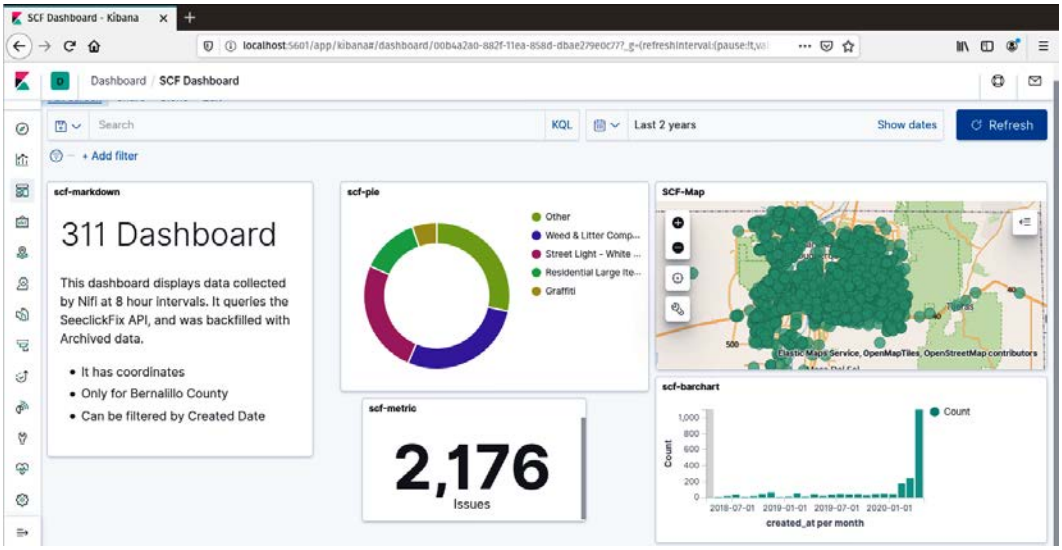

Figure 6.13 – A SeeClickFix dashboard

The dashboard has the Markdown, pie chart, map, metric, and bar chart added. I moved them around by grabbing the top of the panel and resized them by grabbing the lower-right corner and dragging. You can also click the gear icon and add a new name for your panels, so that they do not have the name that you used when you save the visualization.

With your dashboard, you can filter the data and all the visualizations will change. For example, I have clicked on the Graffiti label in the pie chart and the results are shown in the following screenshot:

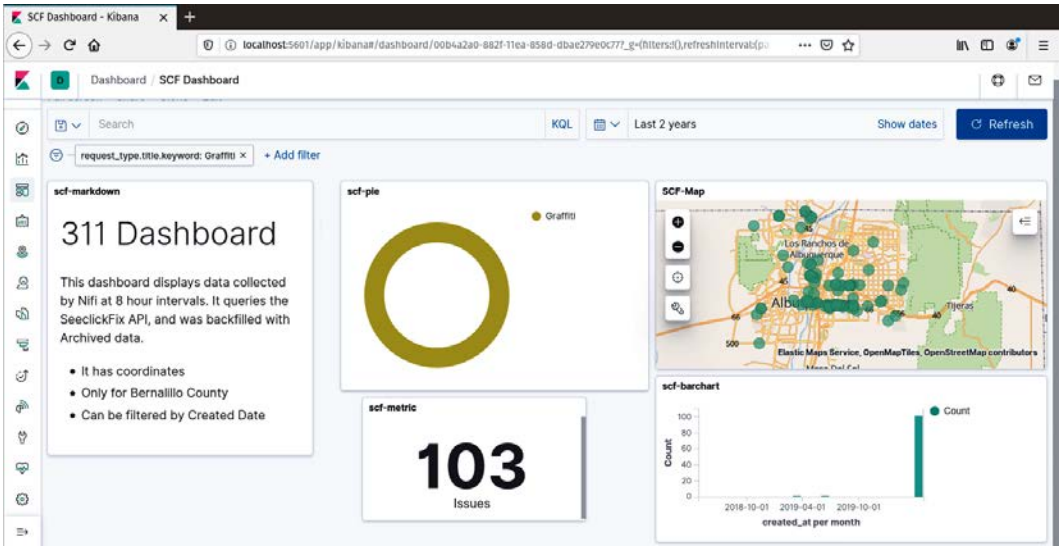

Figure 6.14 – Filtering on Graffiti

Using filters is where the metric visualization comes in handy. It is nice to know what the number of records are. You can see that the map and the bar chart changed as well. You can also filter on the date range. I have selected the last 7 days in the filter, as shown in the following screenshot:

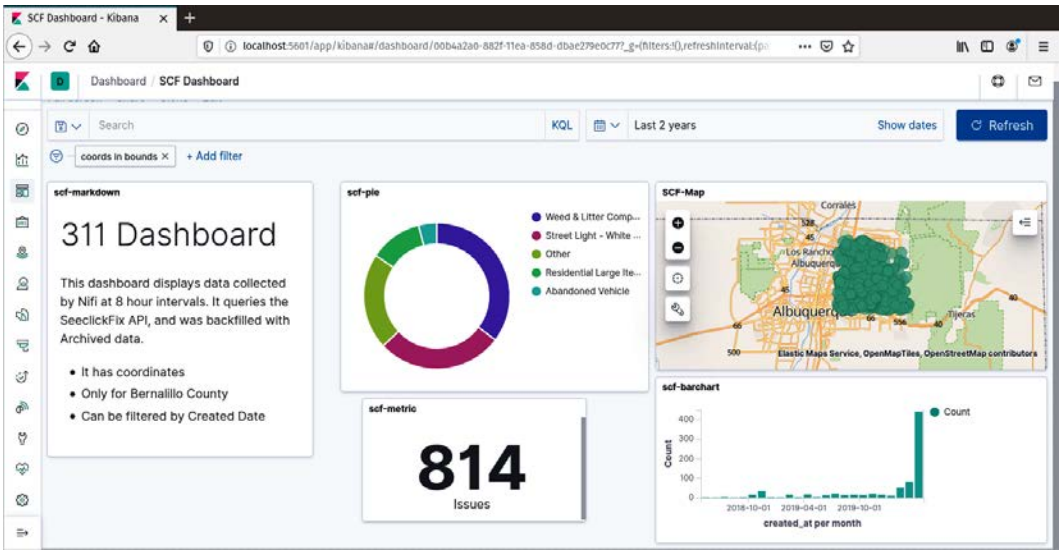

Figure 6.15 – Filtering by time in a dashboard

The time filter allows you to select **Now**, **Relative**, or **Absolute**. **Relative** is a number of days, months, years, and so on from **Now**, while **Absolute** allows you to specify a start and end time on a calendar. The results of the seven-day filter are shown in the following screenshot:

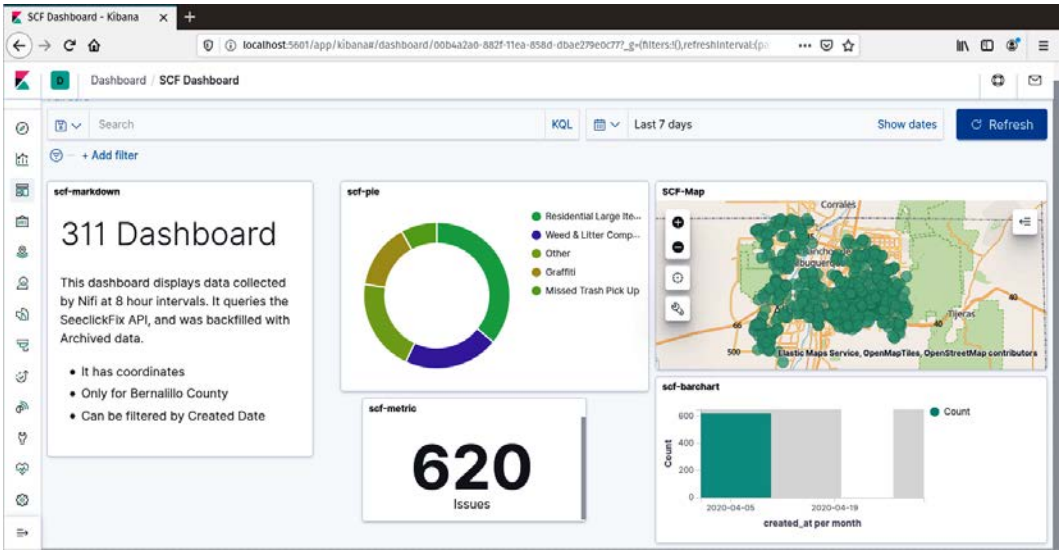

Figure 6.16 – Dashboard with a seven-day filter

The last filter I will show is the map filter. You can select an area or draw a polygon on the map to filter your dashboard. By clicking on the map tools icon, the options will appear as shown in the following screenshot:

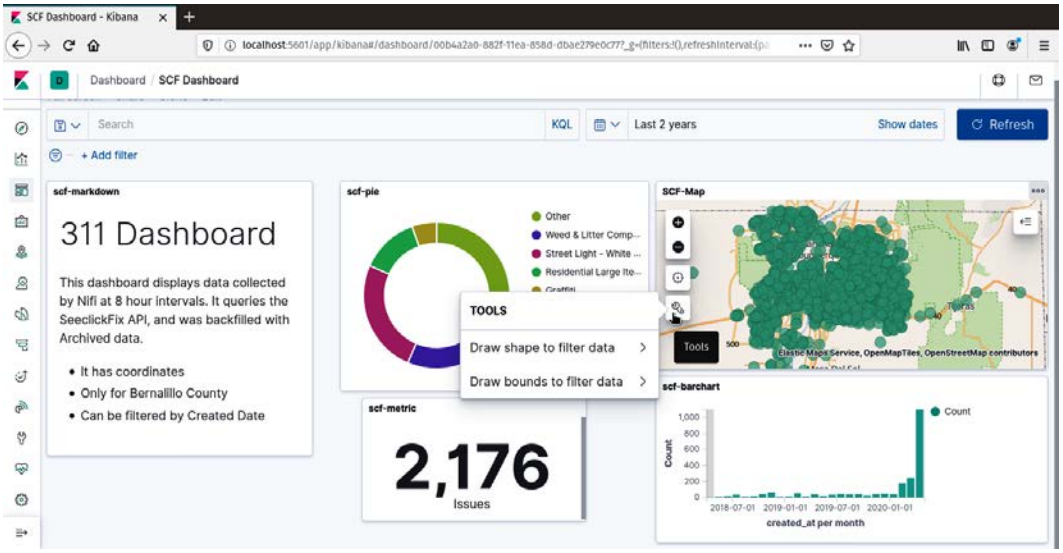

Figure 6.17 – Tools icon on the map

Using the **Draw** bounds to filter data, I drew a rectangle on the map and the results are shown in the following screenshot:

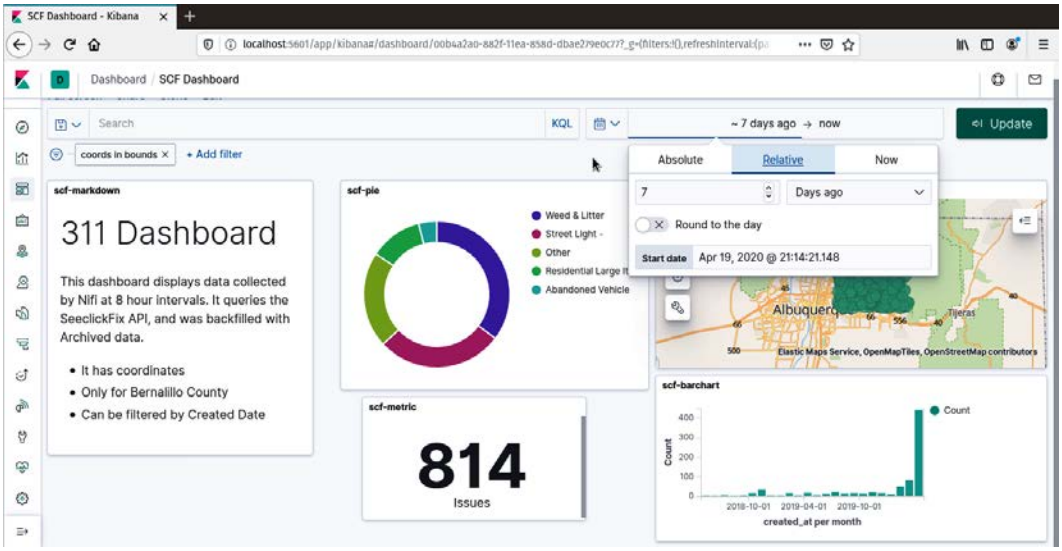

Figure 6.18 – Filtering data using the map

In the preceding dashboard, you can see the perfect rectangle of points. The map filter is one of my favorite filters.

Kibana dashboards make your data pipelines useful to non-data engineers. The work you put into moving and transforming data becomes live data that can be used by analysts and mangers to explore and learn from the data. Kibana dashboards are also an excellent way for you, the data engineer, to visualize the data you have extracted, transformed, and loaded to see whether there are any obvious issues in your data pipeline. They can be a type of debugging tool.

Summary

In this chapter, you learned how to build a data pipeline using data from a REST API. You also added a flow to the data pipeline to allow you to backfill the data, or to recreate a database with all of the data using a single pipeline.

The second half of the chapter provided a basic overview of how to build a dashboard using Kibana. Dashboards will usually be outside the responsibilities of a data engineer. In smaller firms, however, this could very well be your job. Furthermore, being able to quickly build a dashboard can help validate your data pipeline and look for any possible errors in the data.

In the next chapter, we begin a new section of this book, where you will take the skills you have learned and improve them by making your pipelines ready for production. You will learn about deployment, better validation techniques, and other skills needed when you are running pipelines in a production environment.

Section 2: Deploying Data Pipelines in Production

Section 2 builds on what you have learned and teaches you the features of production data pipelines. You will learn techniques that are similar to software engineering, such as versioning, monitoring, and logging. With these skills, you will be able to not only build, but also manage production data pipelines. Lastly, you will learn how to deploy your data pipelines in a production environment.

This section comprises the following chapters:

- *Chapter 7*, *Features of a Production Data Pipeline*
- *Chapter 8*, *Version Control Using the NiFi Registry*
- *Chapter 9*, *Monitoring and Logging Data Pipelines*
- *Chapter 10*, *Deploying Your Data Pipelines*
- *Chapter 11*, *Building a Production Data Pipeline*

7
Features of a Production Pipeline

In this chapter, you will learn several features that make a data pipeline ready for production. You will learn about building data pipelines that can be run multiple times without changing the results (idempotent). You will also learn what to do if transactions fail (atomicity). And you will learn about validating data in a staging environment. This chapter will use a sample data pipeline that I currently run in production.

For me, this pipeline is a bonus, and I am not concerned with errors, or missing data. Because of this, there are elements missing in this pipeline that should be present in a mission critical, or production, pipeline. Every data pipeline will have different acceptable rates of errors – missing data – but in production, your pipelines should have some extra features that you have yet to learn.

In this chapter, we're going to cover the following main topics:

- Staging and validating data
- Building idempotent data pipelines
- Building atomic data pipelines

Staging and validating data

When building production data pipelines, staging and validating data become extremely important. While you have seen basic data validation and cleaning in *Chapter 5, Cleaning, Transforming, and Enriching Data*, in production, you will need a more formal and automated way of performing these tasks. The next two sections will walk you through how to accomplish staging and validating data in production.

Staging data

In the NiFi data pipeline examples, data was extracted, and then passed along a series of connected processors. These processors performed some tasks on the data and sent the results to the next processor. But what happens if a processor fails? Do you start all over from the beginning? Depending on the source data, that may be impossible. This is where staging comes in to play. We will divide staging in to two different types: the staging of files or database dumps, and the staging of data in a database that is ready to be loaded into a warehouse.

Staging of files

The first type of staging we will discuss is the staging of data in files following extraction from a source, usually a transactional database. Let's walk through a common scenario to see why we would need this type of staging.

You are a data engineering at Widget Co – a company that has disrupted widget making and is the only online retailer of widgets. Every day, people from all over the world order widgets on the company website. Your boss has instructed you to build a data pipeline that takes sales from the website and puts them in a data warehouse every hour so that analysts can query the data and create reports.

Since sales are worldwide, let's assume the only data transformation required is the conversion of the local sales date and time to be in GMT. This data pipeline should be straightforward and is shown in the following screenshot:

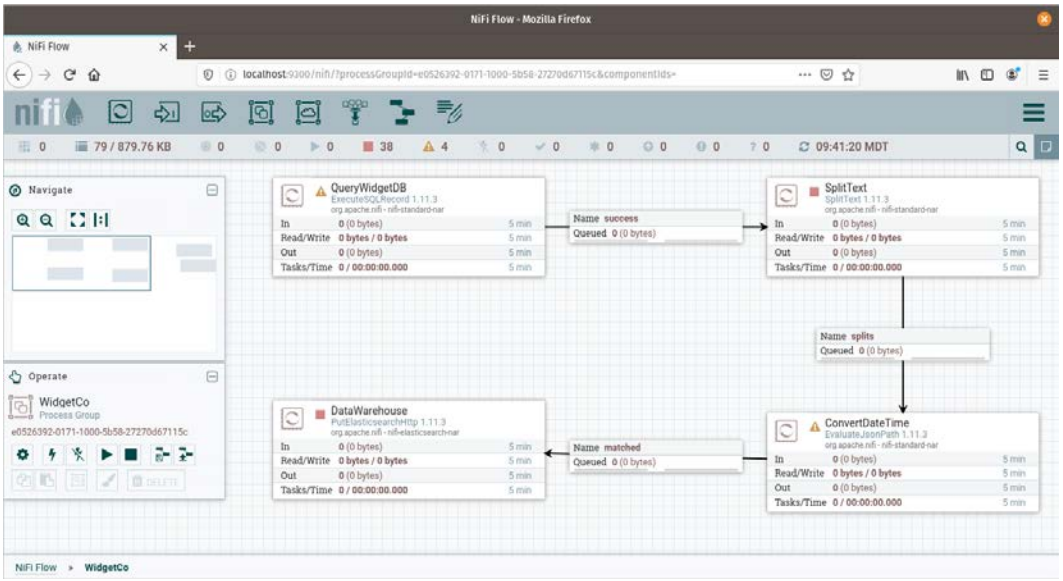

Figure 7.1 – A data pipeline to load widget sales into a warehouse

The preceding data pipeline queries the widget database. It passes the records as a single flowfile to the `SplitText` processor, which sends each record to the processor, which will convert the date and time to GMT. Lastly, it loads the results in the data warehouse.

But what happens when you split the records, and then a date conversion fails? You can just re-query the database, right? No, you can't, because transactions are happening every minute and the transaction that failed was canceled and is no longer in the database, or they changed their order and now want a red widget and not the five blue widgets they initially ordered. Your marketing team will not be happy because they no longer know about these changes and cannot plan for how to convert these sales.

The point of the example is to demonstrate that in a transactional database, transactions are constantly happening, and data is being modified. Running a query produces a set of results that may be completely different if you run the same query 5 minutes later, and you have now lost that original data. This is why you need to stage your extracts.

If the preceding pipeline example is used for staging, you will end up with a pipeline like the example shown in the following screenshot:

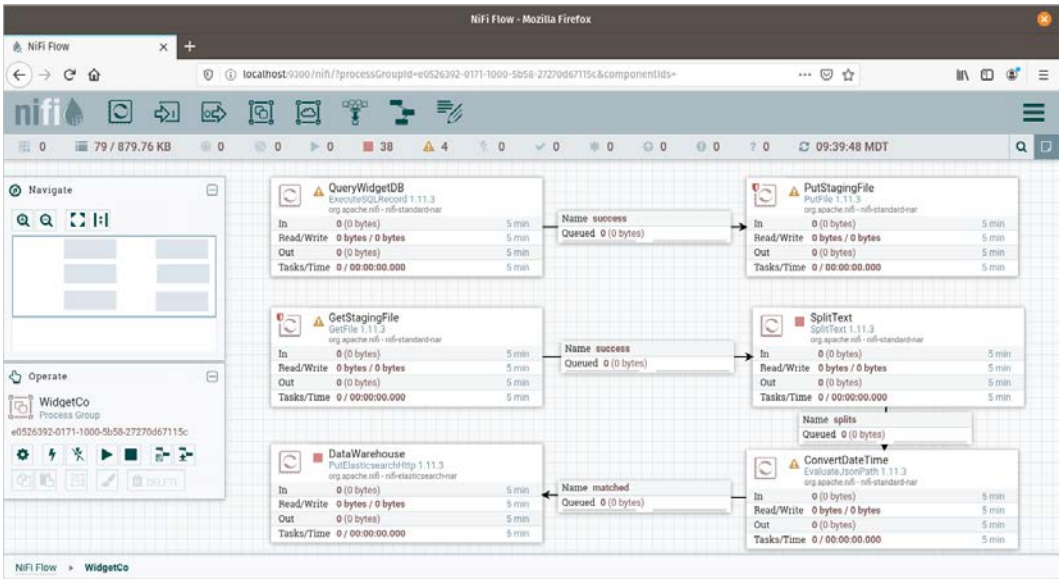

Figure 7.2 – A data pipeline to load widget sales into a warehouse using staging

The preceding data pipeline is displayed as two graphs. The first graph queries the widget database and puts the results in a file on disk. This is the staging step. From here, the next graph will load the data from the staging file, split the records into flowfiles, convert the dates and times, and finally, load it into the warehouse. If this portion of the pipeline crashes, or you need to replay your pipeline for any reason, you can then just reload the CSV by restarting the second half of the data pipeline. You have a copy of the database at the time of the original query. If, 3 months from now, your warehouse is corrupted, you could replay your data pipeline with the data at every query, even though the database is completely different.

Another benefit of having copies of database extracts in CSV files is that it reduces the load in terms of replaying your pipeline. If your queries are resource intensive, perhaps they can only be run at night, or if the systems you query belong to another department, agency, or company. Instead of having to use their resources again to fix a mistake, you can just use the copy.

In the Airflow data pipelines you have built up to this point, you have staged your queries. The way Airflow works encourages good practices. Each task has saved the results to a file, and then you have loaded that file in the next task. In NiFi, however, your queries have been sent, usually to the `SplitRecords` or `Text` processor, to the next processor in the pipeline. This is not good practice for running pipelines in production and will no longer be the case in examples from here on in.

Staging in databases

Staging data in files is helpful during the extract phase of a data pipeline. On the other end of the pipeline, the load stage, it is better to stage your data in a database, and preferably, the same database as the warehouse. Let's walk through another example to see why.

You have queried your data widget database and staged the data. The next data pipeline picks up the data, transforms it, and then loads it into the warehouse. But now what happens if loading does not work properly? Perhaps records went in and everything looks successful, but the mapping is wrong, and dates are strings. Notice I didn't say the load failed. You will learn about handling load failures later in this chapter.

Without actually loading the data into a database, you will only be able to guess what issues you may experience. By staging, you will load the data into a replica of your data warehouse. Then you can run validation suites and queries to see whether you get the results you expect – for example, you could run a `select count(*)` query from the table to see whether you get the correct number of records back. This will allow you to know exactly what issues you may have, or don't have, if all went well.

A data pipeline for Widget Co that uses staging at both ends of the pipeline should look like the pipeline in the following screenshot:

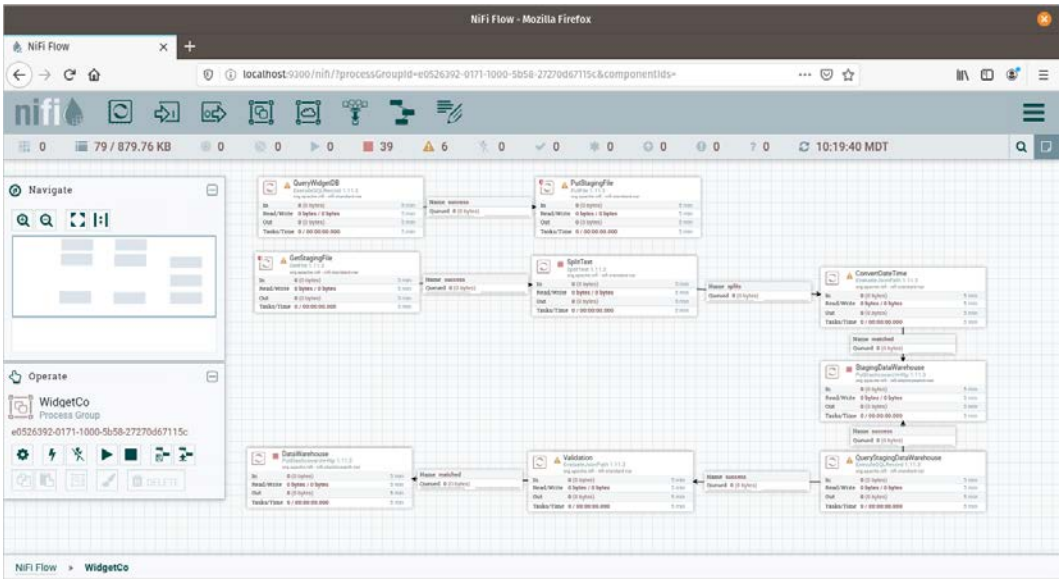

Figure 7.3 – A production using staging at both ends of the pipeline

The data pipeline in the preceding screenshot queries the widget database and stages the results in a file. The next stage picks up the file and converts the dates and times. The point of departure from the earlier example is that the data pipeline now loads the data into a replica of the data warehouse. The new segment of the data pipeline then queries this replica, performs some validation, and then loads it into the final database or warehouse.

ETL versus ELT

So far, you have seen Extract, Transform, and Load. However, there is a growing shift toward an Extract, Load, and Transform process. In the ELT process, data is staged in a database immediately after the extract ion without any transformations. You handle all of the transformations in the database. This is very helpful if you are using SQL-based transformation tools. There is no right or wrong way, only preferences and use cases.

By staging data at the front and end of your data pipeline, you are now better suited for handling errors and for validating the data as it moves through your pipeline. Do not think that these are the only two places where data can be staged, or that data must be staged in files. You can stage your data after every transformation in your data pipeline. Doing so will make debugging errors easier and allow you to pick up at any point in the data pipeline after an error. As your transformations become more time consuming, this may become more helpful.

You staged the extraction from the widget database in a file, but there is no reason to prevent you from extracting the data to a relational or noSQL database. Dumping data to files is slightly less complicated than loading it into a database – you don't need to handle schemas or build any additional infrastructure.

While staging data is helpful for replaying pipelines, handling errors, and debugging your pipeline, it is also helpful in the validation stages of your pipeline. In the next section, you will learn how to use Great Expectations to build validation suites on both file and database staged data.

Validating data with Great Expectations

With your data staged in either a file or a database, you have the perfect opportunity to validate it. In *Chapter 5, Cleaning, Transforming, and Enriching Data*, you used pandas to perform exploratory data analysis and gain insight into what columns existed, find counts of null values, look at ranges of values within columns, and examine the data types in each column. Pandas is powerful and, by using methods such as `value_counts` and `describe`, you can gain a lot of insight into your data, but there are tools that make validation much cleaner and make your expectations of the data much more obvious.

The library you will learn about in this section is **Great Expectations**. The following is a screenshot of the Great Expectations home page, where you can join and get involved with it:

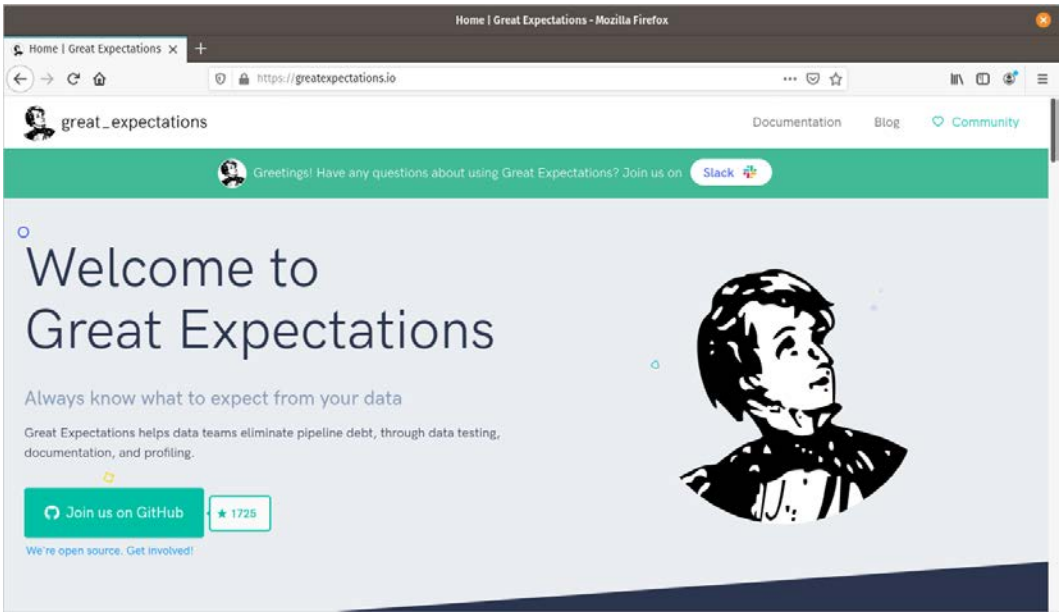

Figure 7.4 – Great Expectations Python library for validating your data, and more

Why Great Expectations? Because with Great Expectations, you can specify human-readable expectations and let the library handle the implementation. For example, you can specify that the `age` column should not have null values, in your code, with the following line:

```
expect_column_values_to_not_be_null('age')
```

Great Expectations will handle the logic behind doing this irrespective of whether your data is in a DataFrame or in a database. The same expectation will run on either data context.

Getting started with Great Expectations

Installing Great Expectations can be done with `pip3` as shown:

```
pip3 install great_expectations
```

To view the documents that Great Expectations generates, you will also need to have Jupyter Notebook available on your machine. You can install Notebook with `pip3` as well:

```
pip3 install jupyter
```

With the requirements installed, you can now set up a project. Create a directory at `$HOME/peoplepipeline` and press *Enter*. You can do this on Linux using the following commands:

```
mkdir $HOME/peoplepipeline
cd $HOME/peoplepipeline
```

Now that you are in the project directory, before you set up Great Expectations, we will dump a sample of the data we will be working with. Using the code from *Chapter 3, Reading and Writing Files*, we will generate 1,000 records relating to people. The code is as follows:

```
from faker import Faker
import csv
output=open('people.csv','w')
fake=Faker()
header=['name','age','street','city','state','zip','lng','lat']
mywriter=csv.writer(output)
mywriter.writerow(header)
for r in range(1000):
    mywriter.writerow([fake.name(),fake.random_int(min=18,
    max=80, step=1), fake.street_address(), fake.city(),fake.
    state(),fake.zipcode(),fake.longitude(),fake.latitude()])
output.close()
```

The preceding code creates a CSV file with records about people. We will put this CSV file into the project directory.

Now you can set up Great Expectations on this project by using the command-line interface. The following line will initialize your project:

```
great_expectations init
```

You will now walk through a series of steps to configure Great Expectations. First, Great Expectations will ask you whether you are ready to proceed. Your terminal should look like the following screenshot:

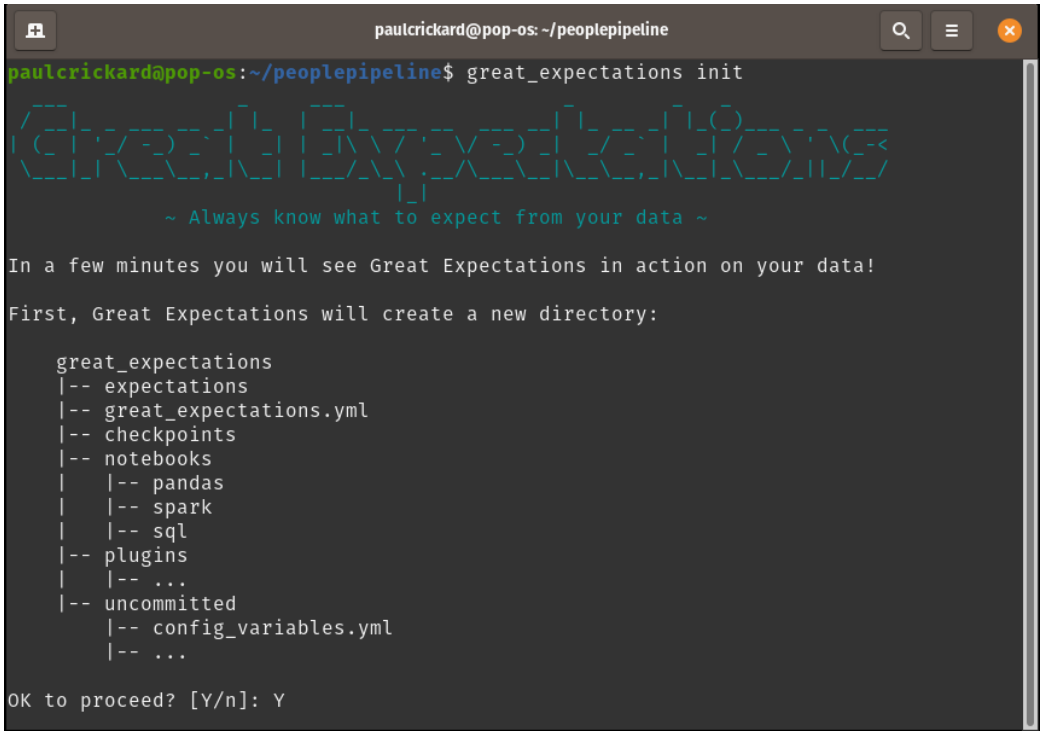

Figure 7.5 – Initializing Great Expectations on a project

Having entered *Y* and pressed *Enter*, you will be prompted with a series of questions:

```
What data would you like Great Expectations to connect
to?
What are you processing your files with?
Enter the path (relative or absolute) of a data file.
Name the new expectation suite [people.warning].
```

The answers to the questions are shown in the following screenshot, but it should be `Files`, `Pandas`, where you put your file, and whatever you would like to name it:

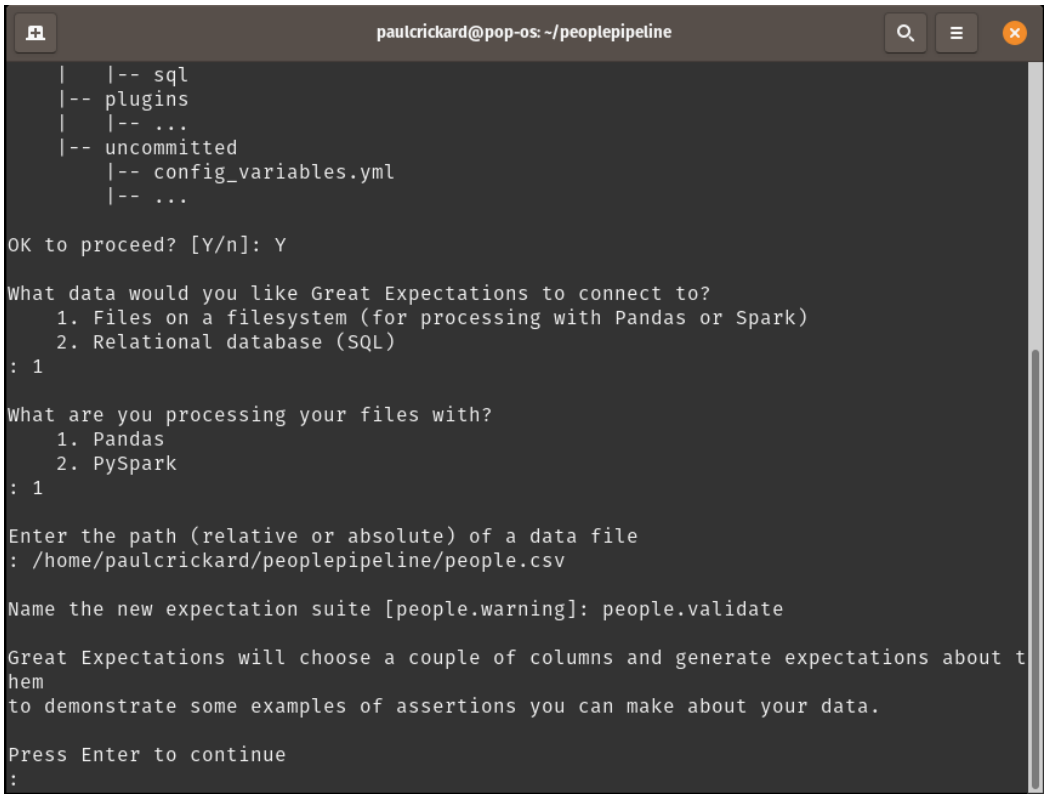

Figure 7.6 – Initializing Great Expectations by answering questions

When Great Expectations has finished running, it will tell you it's done, give you a path to the document it has generated, and open the document in your browser. The documents will look like the following screenshot:

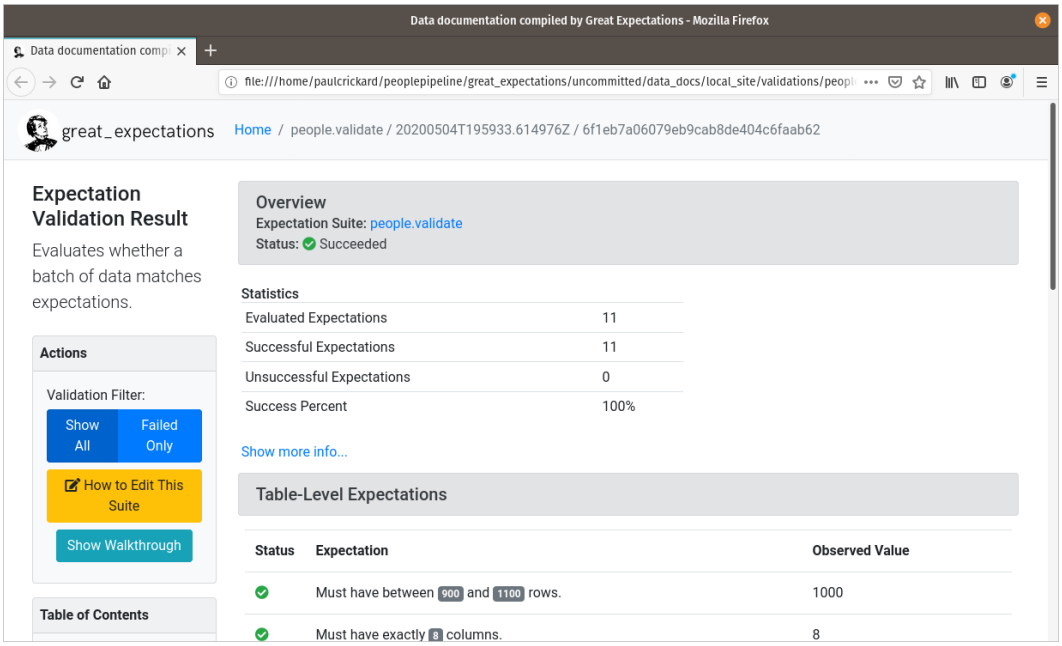

Figure 7.7 – Documentation generated by Great Expectations

The preceding screenshot shows the documentation generated for the Great Expectations Suite. You can see there are **11** expectations and we have passed all of them. The expectations are very basic, specifying how many records should exist and what columns should exist in what order. Also, in the code I specified an age range. So, **age** has a minimum and maximum value. Ages have to be greater than 17 and less than 81 to pass the validation. You can see a sample of the expectations generated by scrolling. I have shown some of mine in the following screenshot:

Status	Expectation	Observed Value
✓	Must have between 900 and 1100 rows.	1000
✓	Must have exactly 8 columns.	8
✓	Must have these columns in this order: name, age, street, city, state, zip, lng, lat	['name', 'age', 'street', 'city', 'state', 'zip', 'lng', 'lat']

age

Status	Expectation	Observed Value
✓	values must never be null.	100% not null
✓	minimum value must be between 17 and 19.	18
✓	maximum value must be between 79 and 81.	80
✓	mean must be between 49.151 and 51.151.	50.151

Figure 7.8 – Sample generated expectations

As you can see, the expectations are very rigid – age must never be null, for example. Let's edit the expectations. You have installed Jupyter Notebook, so you can run the following command to launch your expectation suite in a single step:

```
great_expectations suite edit people.validate
```

Your browser will open a Jupyter notebook and should look like the following screenshot:

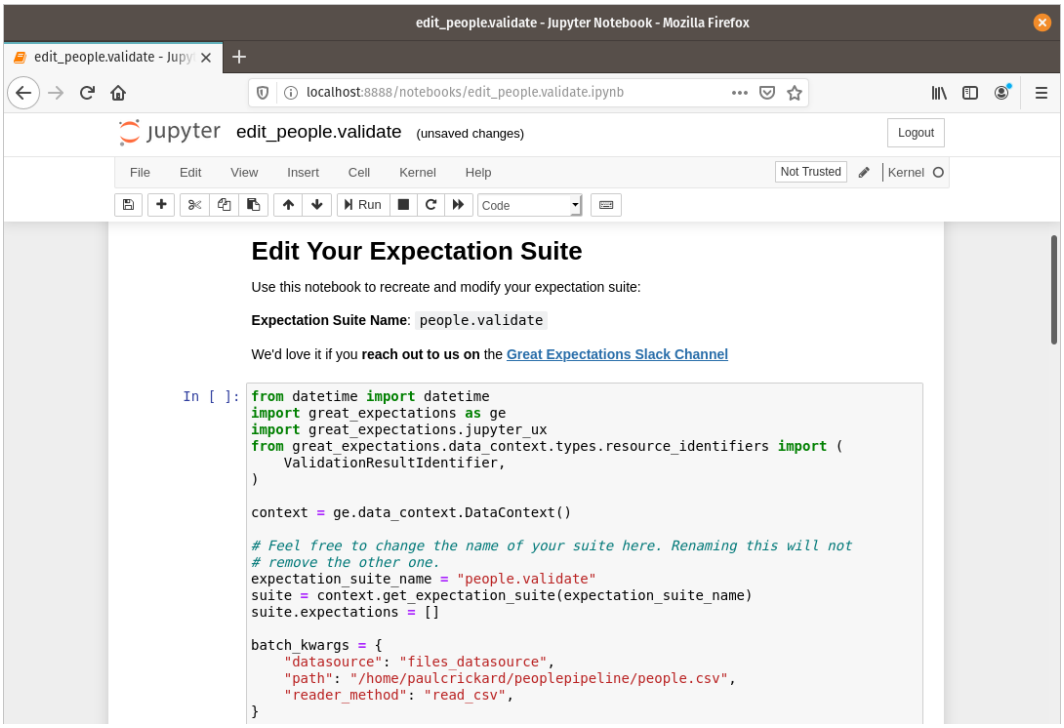

Figure 7.9 – Your expectation suite in a Jupyter notebook

Some items should stand out in the code – the expectation suite name, and the path to your data file in the `batch_kwargs` variable. As you scroll through, you will see the expectations with headers for their type. If you scroll to the `Table_Expectation(s)` header, I will remove the row count expectation by deleting the cell, or by deleting the code in the cell, as shown in the following screenshot:

Table Expectation(s)

```
In [ ]: batch.expect_table_row_count_to_be_between(max_value=1100, min_value=900)

In [ ]: batch.expect_table_column_count_to_equal(value=8)

In [ ]: batch.expect_table_columns_to_match_ordered_list(
            column_list=["name", "age", "street", "city", "state", "zip", "lng", "lat"]
        )
```

Figure 7.10 – Table Expectation(s)

The other expectation to edit is under the `age` header. I will remove an expectation, specifically, the `expect_quantile_values_to_be_between` expectation. The exact line is shown in the following screenshot:

age

```
In [ ]: batch.expect_column_values_to_not_be_null("age")

In [ ]: batch.expect_column_min_to_be_between("age", max_value=19, min_value=17)

In [ ]: batch.expect_column_max_to_be_between("age", max_value=81, min_value=79)

In [ ]: batch.expect_column_mean_to_be_between("age", max_value=51.151, min_value=49.151)

In [ ]: batch.expect_column_median_to_be_between("age", max_value=52.0, min_value=50.0)

In [ ]: batch.expect_column_quantile_values_to_be_between(
            "age",
            quantile_ranges={
                "quantiles": [0.05, 0.25, 0.5, 0.75, 0.95],
                "value_ranges": [[21, 23], [34, 36], [50, 52], [64, 66], [76, 78]],
            },
        )
```

Figure 7.11 – Age expectations with the quantile expectations to be removed

You can continue to remove expectations, or you can add new ones, or even just modify the values of existing expectations. You can find a glossary of available expectations at `https://docs.greatexpectations.io/en/latest/reference/glossary_of_expectations.html`.

Once you have made all of the changes and are satisfied, you can run the entire notebook to save the changes to your expectation suite. The following screenshot shows how to do that – select **Cell | Run All**:

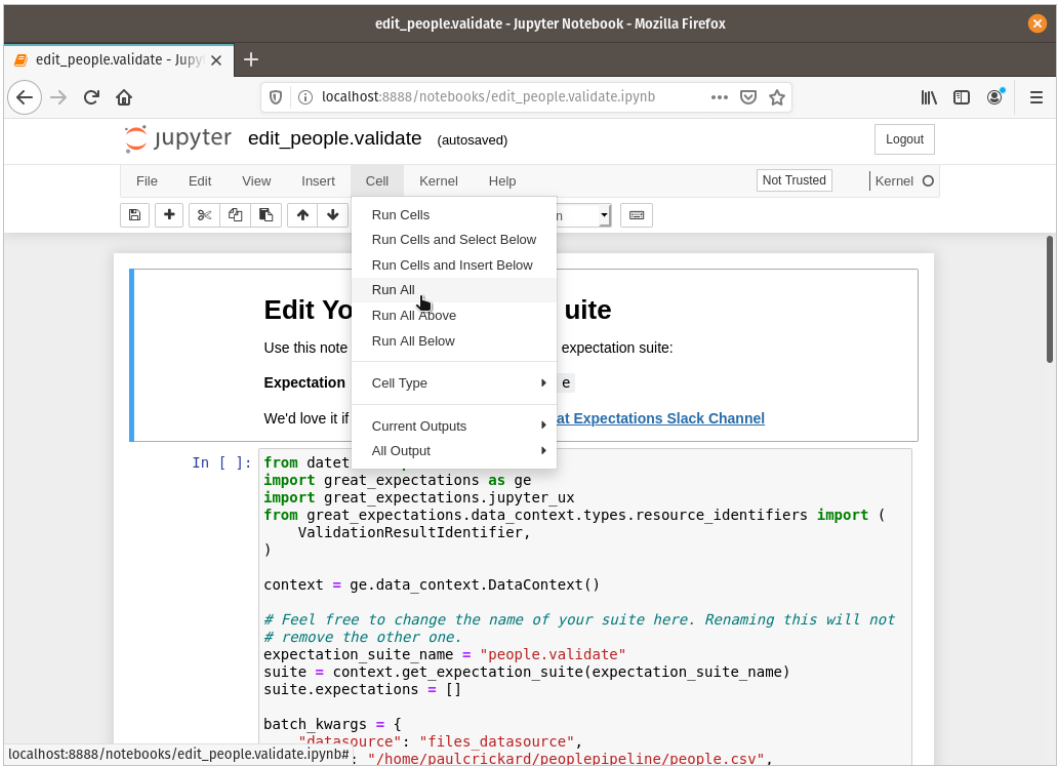

Figure 7.12 – Saving the changes to your expectation suite by running the notebook

Now that you have an expectation suite, it is time to add it to your pipeline. In the next two sections, you will learn how to add it alongside your pipeline for use with NiFi or embed the code into your pipeline for use with Airflow.

Great Expectations outside the pipeline

So far, you have validated data while you edited the expectation suite inside a Jupyter notebook. You could continue to do that using a library such as Papermill, but that is beyond the scope of this book. In this section, however, you will create a tap and run it from NiFi.

Papermill

Papermill is a library created at Netflix that allows you to create parameterized Jupyter notebooks and run them from the command line. You can change parameters and specify an output directory for the resultant notebook. It pairs well with another Netflix library, Scrapbook. Find them both, along with other interesting projects, including Hydrogen, at `https://github.com/nteract`.

A tap is how Great Expectations creates executable Python files to run against your expectation suite. You can create a new tap using the command-line interface, as shown:

```
great_expectations tap new people.validate
peoplevalidatescript.py
```

The preceding command takes an expectation suite and the name of a Python file to create. When it runs, it will ask you for a data file. I have pointed it to the `people.csv` file that you used in the preceding section when creating the suite. This is the file that the data pipeline will overwrite as it stages data:

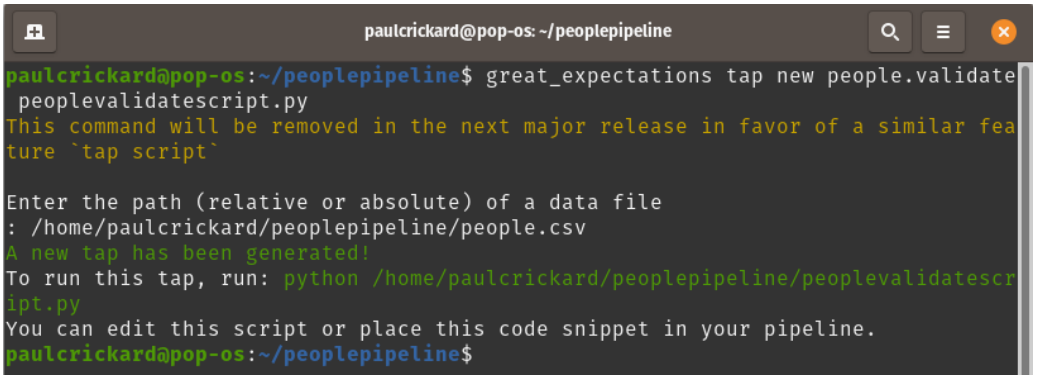

Figure 7.13 – Result of the Python file at the specified location

If you run the tap, you should see that it succeeded, as shown in the following screenshot:

```
paulcrickard@pop-os:~/peoplepipeline$ python3 peoplevalidatescript.py
Validation Succeeded!
```

Figure 7.14 – Great Expectation tap run

You are now ready to build a pipeline in NiFi and validate your data using Great Expectations. The next section will walk you through the process.

Great Expectations in NiFi

Combining NiFi and Great Expectations requires a few modifications to the tap you created in the previous section. First, you will need to change all the exits to be `0`. If you have a `system.exit(1)` exit, NiFi processors will crash because the script failed. We want the script to close successfully, even if the results are not, because the second thing you will change are the `print` statements. Change the `print` statements to be a JSON string with a result key and a pass or fail value. Now, even though the script exits successfully, we will know in NiFi whether it actually passed or failed. The code of the tap is shown in the following code block, with the modifications in bold:

```
import sys
from great_expectations import DataContext
context = DataContext("/home/paulcrickard/peoplepipeline/great_
expectations")
suite = context.get_expectation_suite("people.validate")
batch_kwargs = {
    "path": "/home/paulcrickard/peoplepipeline/people.csv",
    "datasource": "files_datasource",
    "reader_method": "read_csv",
}
batch = context.get_batch(batch_kwargs, suite)
results = context.run_validation_operator(
                                "action_list_operator", [batch])
if not results["success"]:
    print('{"result":"fail"}')
    sys.exit(0)

print('{"result":"pass"}')
sys.exit(0)
```

With the changes to the tap complete, you can now build a data pipeline in NiFi. The following screenshot is the start of a data pipeline using the tap:

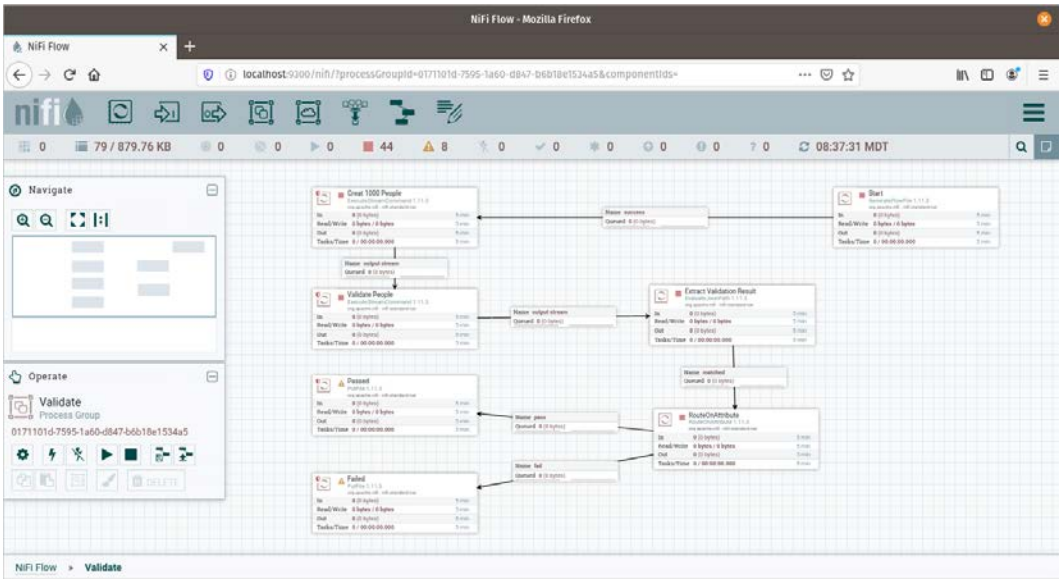

Figure 7.15 – A NiFi data pipeline using Great Expectations

The preceding data pipeline creates 1,000 records and saves it as a CSV file. It then runs the tap on the data and reads in the result — the pass or fail JSON from the script. Lastly, it extracts the result and routes the flowfile to either a pass or fail processor. From there, your data pipeline can continue, or it can log the error. You will walk through the pipeline in the following steps:

1. The data pipeline starts by generating a fake flowfile without any data to trigger the next processor. You could replace this processor with one that queries your transactional database, or that reads files from your data lake. I have scheduled this processor to run every hour.
2. Once the empty flowfile is received, the `ExecuteStreamCommand` processor calls the `loadcsv.py` Python script. This file is from *Chapter 3, Reading and Writing Files*, and uses `Faker` to create 1,000 fake people records. The `ExecuteStreamCommand` processor will read the output from the script. If you had print statements, each line would become a flowfile. The script has one output, and that is `{"status":"Complete"}`.

3. To configure the processor to run the script, you can set the **Working Directory** to the path of your Python script. Set **Command Path** to `python3` – if you can run the command with the full path, you do not need to enter it all. Lastly, set **Command Arguments** to the name of the Python file – `loadcsv.py`. When the processor runs, the output flowfile is shown in the following screenshot:

Figure 7.16 – The flowfile shows the JSON string

4. The next processor is also an `ExecuteStreamCommand` processor. This time, the script will be your tap. The configuration should be the same as in the previous step, except **Command Argument** will be `peoplevalidatescript.py`. Once the processor completes, the flowfile will contain JSON with a result of pass or fail. The `pass` flowfile is shown in the following screenshot:

Figure 7.17 – Result of the tap, validation passed

5. The value of the result is extracted in the next processor – `EvaluateJsonPath`. Adding a new property with the plus button, name it `result` and set the value to `$.result`. This will extract the `pass` or `fail` value and send it as a flowfile attribute.

6. The next process is `RouteOnAttribute`. This processor allows you to create properties that can be used as a relationship in a connection to another processor, meaning you can send each property to a different path. Creating two new properties – `pass` and `fail`, the values are shown in the following code snippet:

```
${result:startsWith('pass')}
${result:startsWith('fail')}
```

7. The preceding command uses the NiFi expression language to read the value of the result attribute in the flowfile.
8. From here, I have terminated the data pipeline at a `PutFile` processor. But you would now be able to continue by connecting a `pass` and `fail` path to their respective relationships in the previous processor. If it passed, you could read the staged file and insert the data into the warehouse.

In this section, you connected Great Expectations to your data pipeline. The tap was generated using your data, and because of this, the test passed. The pipeline ended with the file being written to disk. However, you could continue the data pipeline to route success to a data warehouse. In the real world, your tests will fail on occasion. In the next section, you will learn how to handle failed tests.

Failing the validation

The validation will always pass because the script we are using generates records that meet the validations rules. What if we changed the script? If you edit the `loadcsv.py` script and change the minimum and maximum age, we can make the validation fail. The edit is shown as follows:

```
fake.random_int(min=1, max=100, step=1)
```

This will create records that are below the minimum and above the maximum—hopefully, because it is random, but 1,000 records should get us there. Once you have edited the script, you can rerun the data pipeline. The final flowfile should have been routed to the `fail` path. Great Expectations creates documents for your validations. If you remember, you saw them initially when you created the validation suite. Now you will have a record of both the passed and failed runs. Using your browser, open the documents. The path is within your project folder. For example, my docs are at the following path:

```
file:///home/paulcrickard/peoplepipeline/great_expectations/
uncommitted/data_docs/local_site/validations/people/
validate/20200505T145722.862661Z/6f1eb7a06079eb9cab8de404c6faa
b62.html
```

The documents should show all your validations runs. The documents will look like the following screenshot:

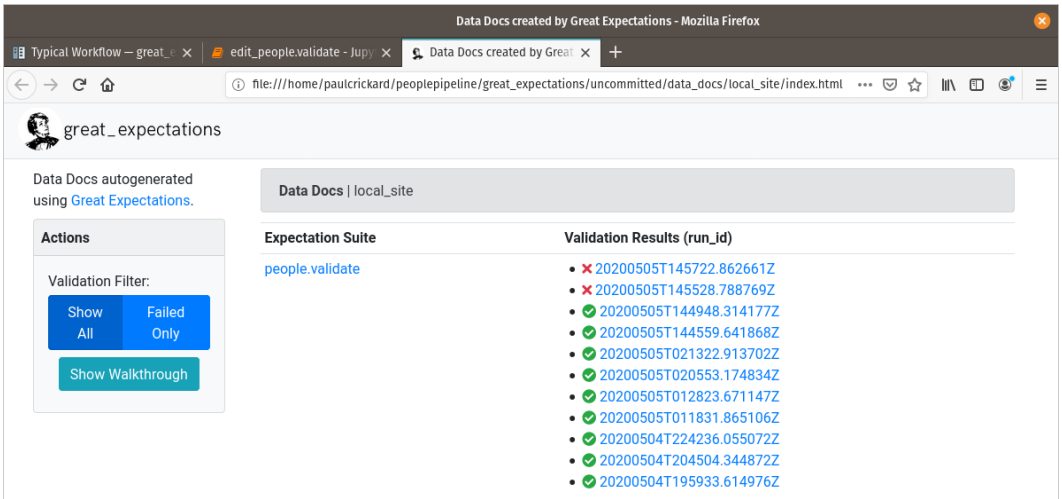

Figure 7.18 – Results of multiple validation runs

The preceding screenshot shows all of the validation runs. You can see the red **x** indicating failures. Click on one of the failed runs to see which expectations were not met. The results should be that both the minimum and maximum age were not met. You should see that this is the case, as shown in the following screenshot:

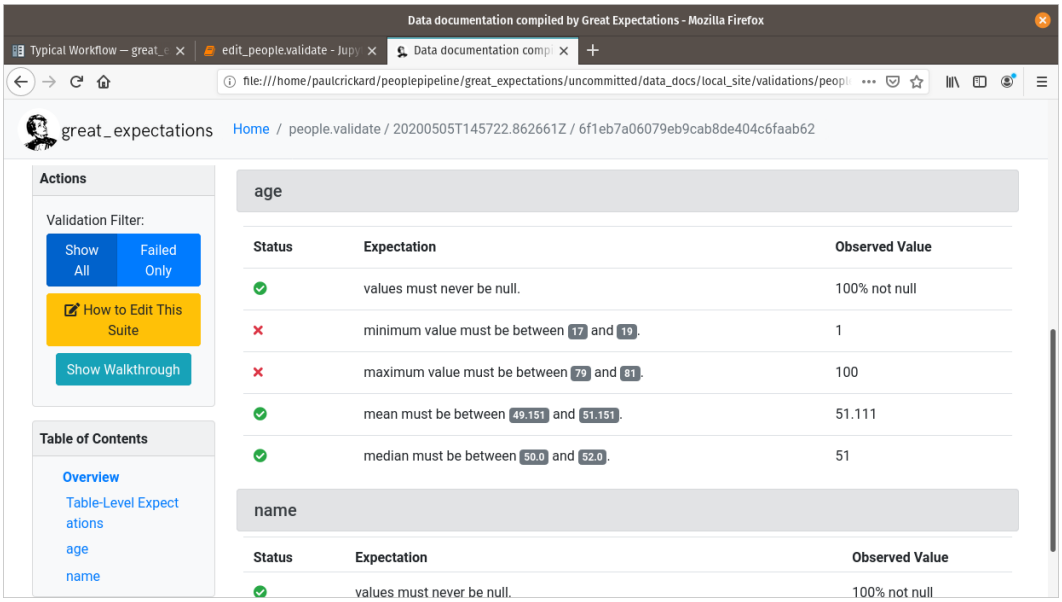

Figure 7.19 – Age expectations have not been met

In this section, you have created a Great Expectations suite and specified expectations for your data. Previously, you would have had to do this manually using DataFrames and a significant amount of code. Now you can use human-readable statements and allow Great Expectations to do the work. You have created a tap that you can run inside your NiFi data pipeline — or that you can schedule using Cron or any other tool.

A quick note on Airflow

In the preceding example, you ran the validation suite outside of your pipeline – the script ran in the pipeline, but was called by a processor. You can also run the code inside the pipeline without having to call it. In Apache Airflow, you can create a validation task that has the code from the tap. To handle the failure, you would need to raise an exception. To do that, import the library in your Airflow code. I have included the libraries that you need to include on top of your standard boilerplate in the following code block:

```
import sys
from great_expectations import DataContext
from airflow.exceptions import AirflowException
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
```

After importing all of the libraries, you can write your task, as shown in the following code block:

```
def validateData():
     context = DataContext("/home/paulcrickard/peoplepipeline/
great_expectations")
     suite = context.get_expectation_suite("people.validate")

     batch_kwargs = {
     "path": "/home/paulcrickard/peoplepipeline/people.csv",
     "datasource": "files_datasource",
     "reader_method": "read_csv",
}

     batch = context.get_batch(batch_kwargs, suite)
     results = context.run_validation_operator(
                                "action_list_operator", [batch])

     if not results["success"]:
           raise AirflowException("Validation Failed")
```

The preceding code will throw an error, or it will end if the validation succeeded. However, choosing to handle the failure is up to you. All you need to do is check whether `results["success"]` is `True`. You can now code the other functions, create the tasks using `PythonOperator`, and then set the downstream relationships as you have in all the other Airflow examples.

The following sections will discuss two other features of a production data pipeline – idempotence and atomicity.

Building idempotent data pipelines

A crucial feature of a production data pipeline is that it is idempotent. Idempotent is defined as *denoting an element of a set that is unchanged in value when multiplied or otherwise operated on by itself*.

In data science, this means that when your pipeline fails, which is not a matter of *if*, but *when*, it can be rerun and the results are the same. Or, if you accidently click run on your pipeline three times in a row by mistake, there are not duplicate records – even if you accidently click run multiple times in a row.

In *Chapter 3, Reading and Writing Files*, you created a data pipeline that generated 1,000 records of people and put that data in an Elasticsearch database. If you let that pipeline run every 5 minutes, you would have 2,000 records after 10 minutes. In this example, the records are all random and you may be OK. But what if the records were rows queried from another system?

Every time the pipeline runs, it would insert the same records over and over again. How you create idempotent data pipelines depends on what systems you are using and how you want to store your data.

In the SeeClickFix data pipeline from the previous chapter, you queried the SeeClickFix API. You did not specify any rolling time frame that would only grab the most recent records, and your backfill code grabbed all the archived issues. If you run this data pipeline every 8 hours, as it was scheduled, you will grab new issues, but also issues you already have.

The SeeClickFix data pipeline used the `upsert` method in Elasticsearch to make the pipeline idempotent. Using the `EvaluteJsonPath` processor, you extracted the issue ID and then used that as the `Identifier Attribute` in the `PutElasticsearchHttp` processor. You also set the **Index Operation** to `upsert`. This is the equivalent of using an update in SQL. No records will be duplicated, and records will only be modified if there have been changes.

Another way to make the data pipeline idempotent, and one that is advocated by some functional data engineering advocates, is to create a new index or partition every time your data pipeline is run. If you named your index with the datetime stamped as a suffix, you would get a new index with distinct records every time the pipeline runs. This not only makes the data pipeline idempotent; it creates an immutable object out of your database indexes. An index will never change; just new indexes will be added.

Building atomic data pipelines

The final feature of a production data pipeline that we will discuss in this chapter is atomicity. Atomicity means that if a single operation in a transaction fails, then all of the operations fail. If you are inserting 1,000 records into the database, as you did in *Chapter 3, Reading and Writing Files*, if one record fails, then all 1,000 fail.

In SQL databases, the database will roll back all the changes if record number 500 fails, and it will no longer attempt to continue. You are now free to retry the transaction. Failures can occur for many reasons, some of which are beyond your control. If the power or the network goes down while you are inserting records, do you want those records to be saved to the database? You would then need to determine which records in a transaction succeeded and which failed and then retry only the failed records. This would be much easier than retrying the entire transaction.

In the NiFi data pipelines you have built, there was no atomicity. In the SeeClickFix example, each issue was sent as a flowfile and upserted in Elasticsearch. The only atomicity that existed is that every field in the document (issue) succeeded or failed. But we could have had a situation where all the issues failed except one, and that would have resulted in the data pipeline succeeding.

Elasticsearch does not have atomic transactions, so any data pipeline that implements Elasticsearch would need to handle that within the logic. For example, you could track every record that is indexed in Elasticsearch as well as every failure relationship. If there is a failure relationship during the run, you would then delete all the successfully indexed issues. An example data pipeline is shown in the following screenshot:

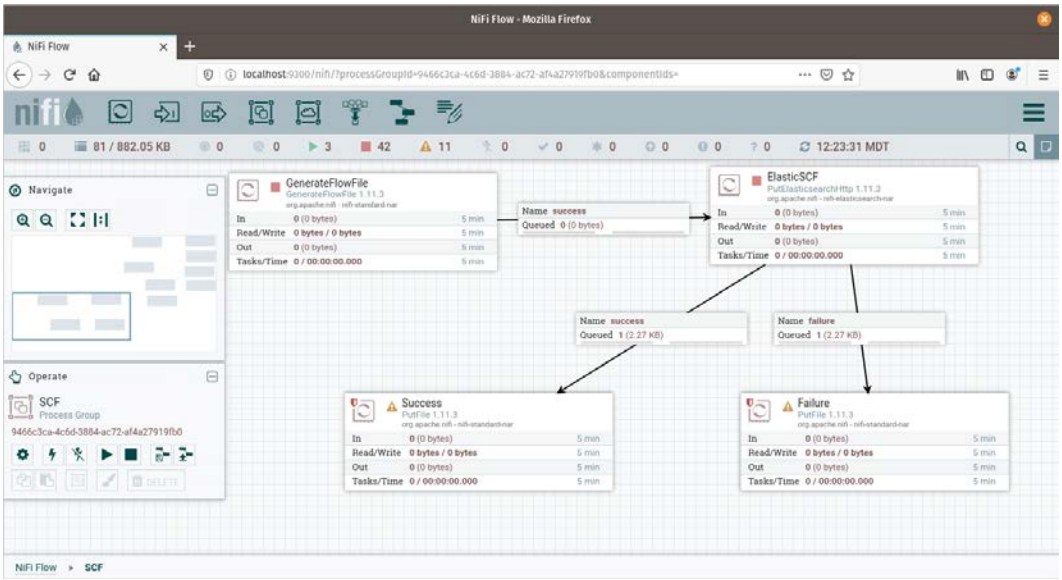

Figure 7.20 – Building atomicity into a data pipeline

The preceding data pipeline created two flowfiles; one succeeded and one failed. The contents of both are put in files on disk. From here, your data pipeline could list the files in the failed directory. If there was one or more, it could then read the success files and remove them from Elasticsearch.

This is not elegant, but atomicity is important. Debugging data pipeline failures when the failure is only partial is extremely difficult and time consuming. The extra work required to incorporate atomicity is well worth it.

SQL databases have atomicity built into the transactions. Using a library such as `psycopg2`, you can roll multiple inserts, updates, or deletes into a single transaction and guarantee that the results will either be that all operations were successful, or the transaction failed.

Creating data pipelines that are idempotent and atomic requires additional work when creating your data pipeline. But without these two features, you will have data pipelines that will make changes to your results if accidently run multiple times (not idempotent) or if there are records that are missing (not atomic). Debugging these issues is difficult, so the time spent on making your data pipelines idempotent and atomic is well spent.

Summary

In this chapter, you learned three key features of production data pipelines: staging and validation, idempotency, and atomicity. You learned how to use Great Expectations to add production-grade validation to your data pipeline staged data. You also learned how you could incorporate idempotency and atomicity into your pipelines. With these skills, you can build more robust, production-ready pipelines.

In the next chapter, you will learn how to use version control with the NiFi registry.

Monitoring NiFi with the status bar

Much of the information you need is on the status bar. The status bar is below the component toolbar and looks like the following screenshot:

Figure 9.1 – Component and status toolbars

Starting at the left of the status bar, let's look at what is being monitored:

- **Active thread**: This lets you know how many threads are running. You can get a sense of tasks and load.
- **Total queued data**: The number of flowfiles and the combined size on disk.
- **Transmitting remote process groups and not transmitting remote process groups**: You can run NiFi on multiple machines or instances on the same machine and allow process groups to communicate. These icons tell you whether they are or are not communicating.
- **Running components, stopped components, invalid components, and disabled components**: These show you the state of your components. Running does not necessarily mean that a component is currently processing data, but that it is on and scheduled to do so.
- **Up-to-date versioned process groups, locally modified versioned process groups, stale versioned process groups, locally modified and stale versioned process groups, and sync failure versioned process groups**: This group of icons show the versioning information of your processor groups. From here you can tell if you have uncommitted changes or are using older versions.
- **Last refresh**: This lets you know when the data in the toolbar is valid for. The refresh is usually every five minutes.

The status bar gives you the monitoring information for all of your processors, but there is also a status toolbar on every processor group and for each processor. You can see the status of the same metrics in the **SCF** processor group, as shown in the following screenshot:

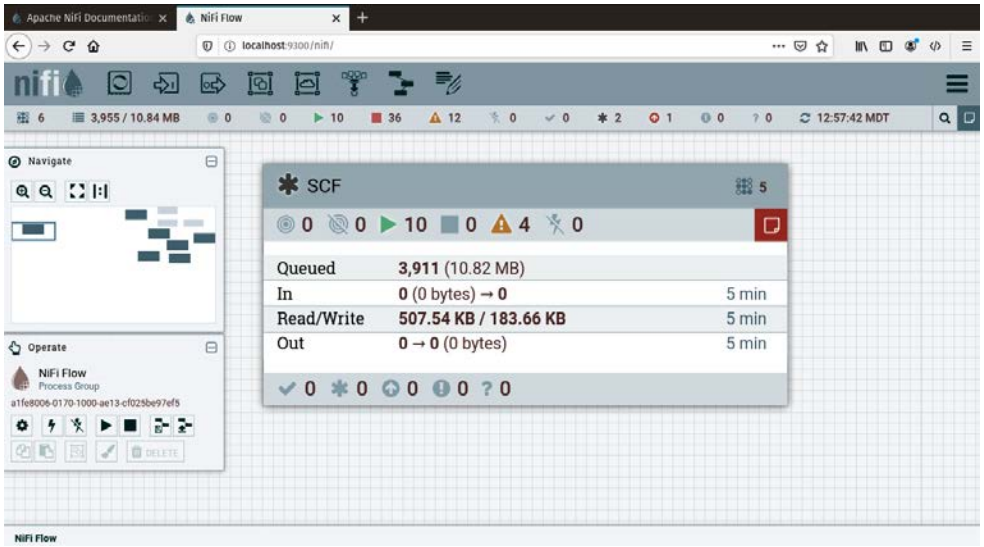

Figure 9.2 – Processor group monitoring

The **In** and **Out** metrics show if there is data flowing into the process group from another processor or group. You will learn how to connect processor groups in the next chapter. The versioning information is not on the toolbar but to the left of the title of the processor group. The red square on the right of the processor group is a **bulletin**. This provides information on errors within the processor group. Hovering over it shows the error, as shown in the following screenshot:

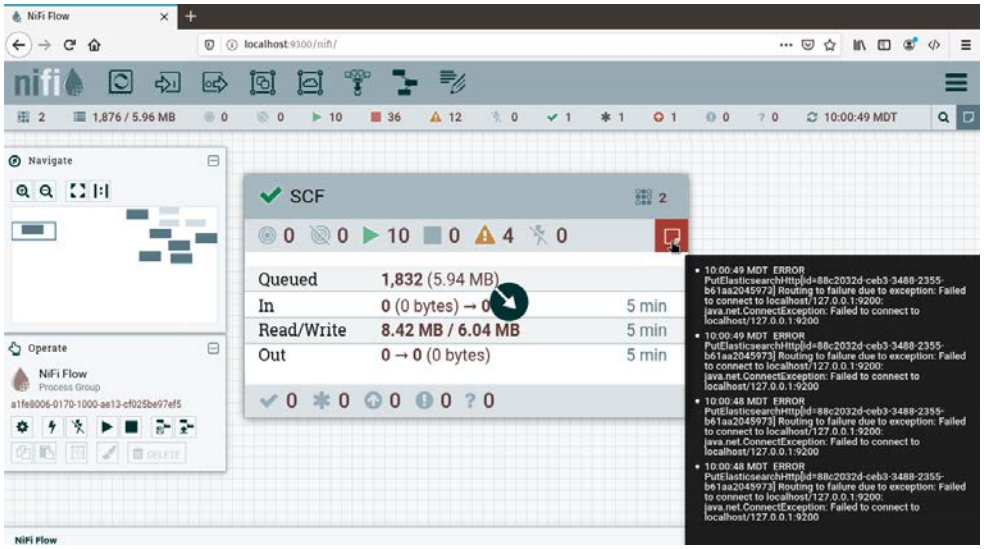

Figure 9.3 – Looking at the bulletin on a processor group

I currently do not have Elasticsearch running and as a result, the processor that sends data to Elasticsearch is failing with a connection timeout. If you enter the processor group, you can see the bulletin on the specific processor, as shown in the following screenshot:

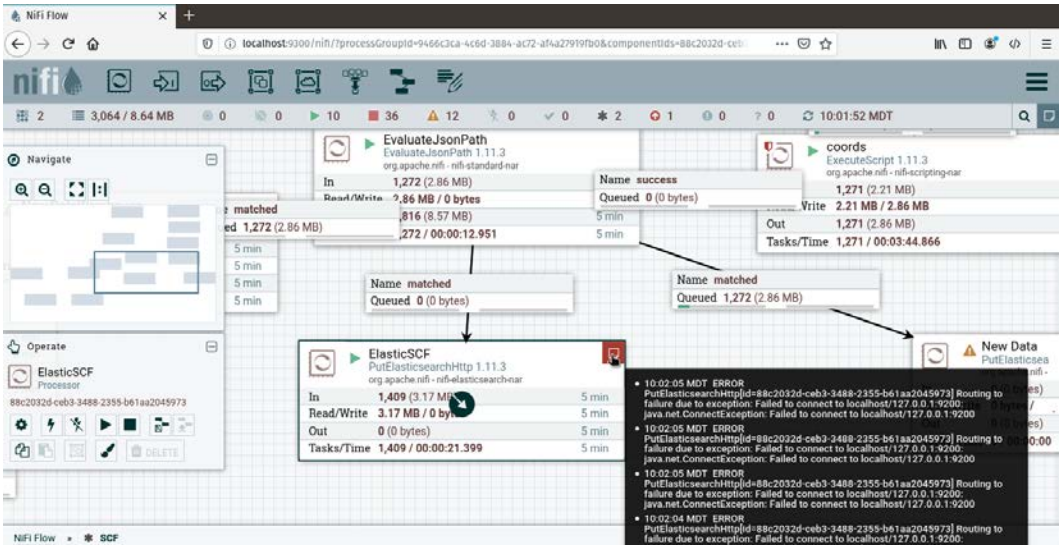

Figure 9.4 – The bulletin on a specific processor

To change the bulletin messages, you can adjust the level in the processor configuration under **Settings**. The **Bulletin Level** dropdown allows you to show more or less based on severity, as shown in the following screenshot:

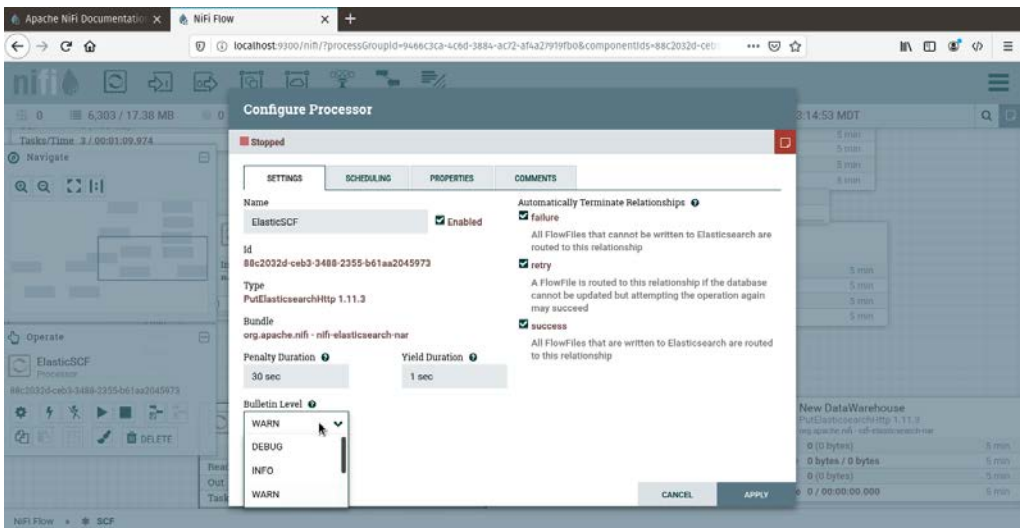

Figure 9.5 – Setting the Bulletin Level

You can see the bulletin information for all your NiFi processors using the **Bulletin Board**, which is accessed from the waffle menu in the upper-right corner of NiFi. Selecting the **Bulletin Board** will show all the messages, as shown in the following screenshot:

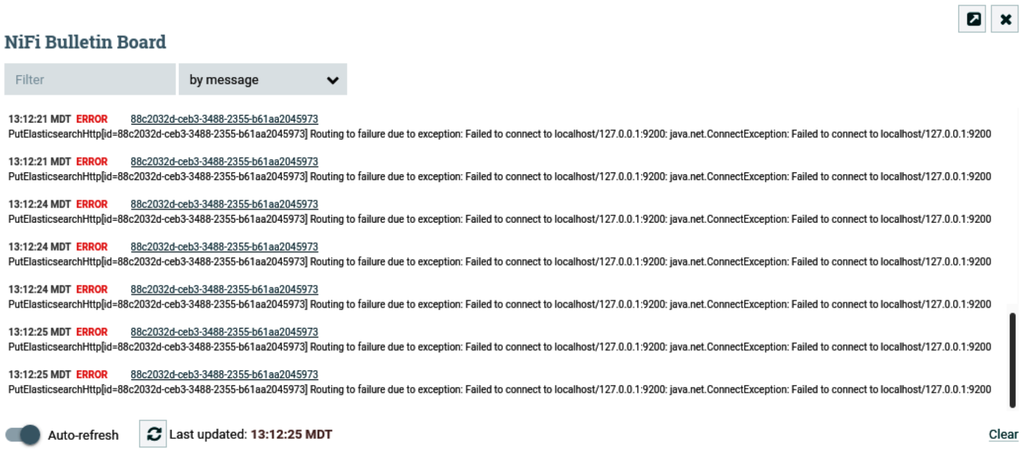

Figure 9.6 – Bulletin Board showing all the notices

Within each processor group, every processor also has status information, as shown in the following screenshot:

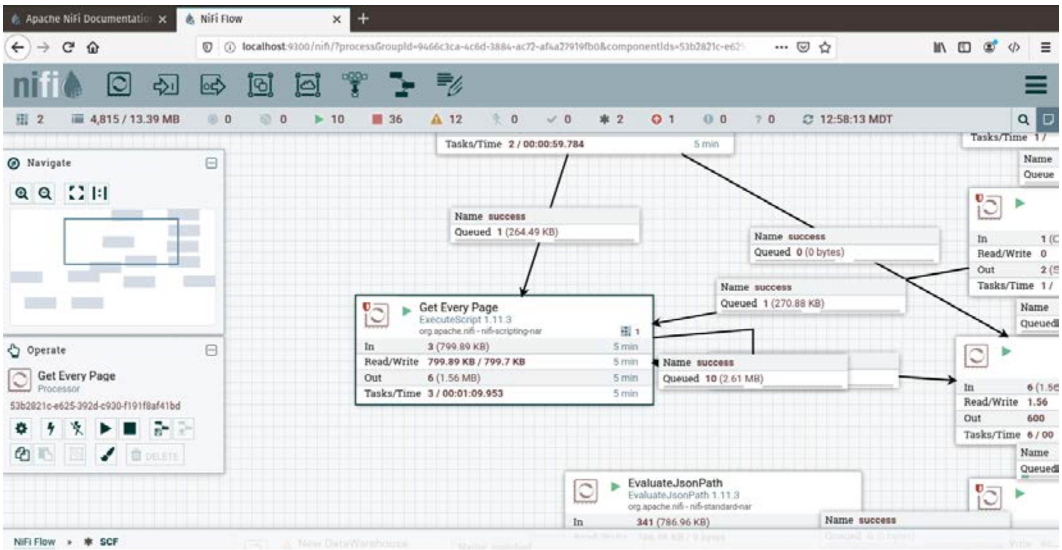

Figure 9.7 – Status of a single processor

The **In** and **Out** metrics in a processor show how much data (the flowfiles size) has passed through the processor in the last five minutes.

Using counters

Similar to bulletins, you can create increment- or decrement-counters. Counters don't tell you that something succeeded or failed, but they can give you an idea of how many flowfiles are being processed at any point in a data pipeline.

In the **SCF** processor group, I have inserted an `UpdateCounter` processor between the `EvaluateJsonPath` and `ElasticSCF` processors. This means that before a flowfile is inserted into Elasticsearch, the counter will be updated. The flow is shown in the following screenshot:

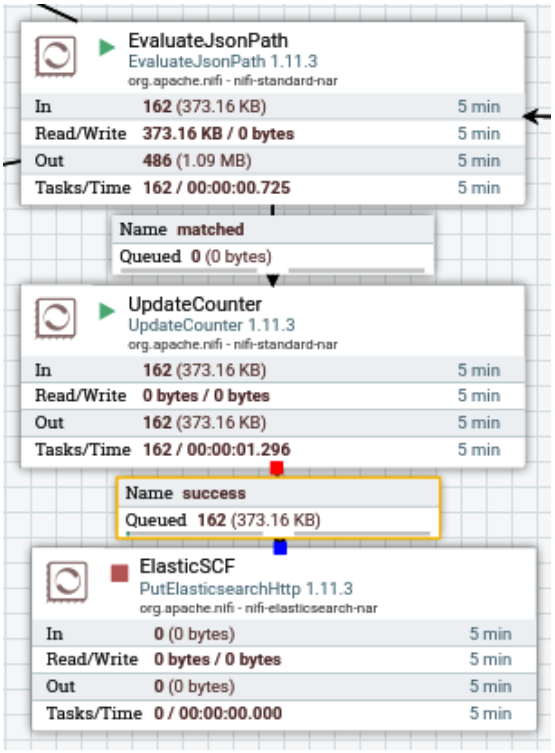

Figure 9.8 – The UpdateCounter processor added to the data pipeline

As you can see in the preceding screenshot, 162 flowfiles were sent through the processor. You will see the results of this later in this section. But first, to configure the processor, you will need to specify **Counter Name** and **Delta**. **Delta** is the number to increment or decrement by. I have configured the processor as shown in the following screenshot:

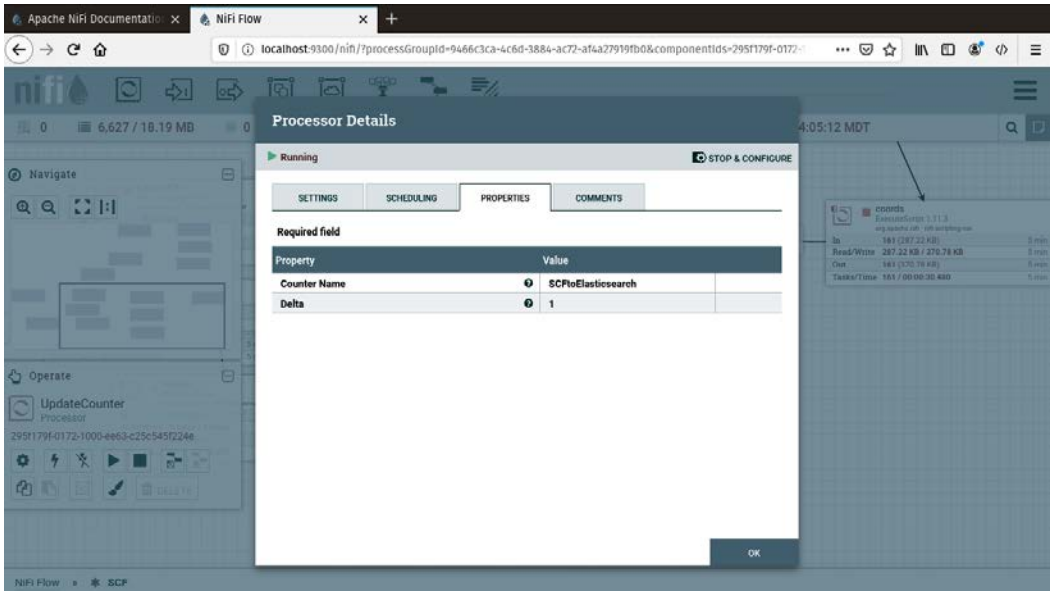

Figure 9.9 – Configuration of the UpdateCounter processor

When you have configured the processor and ran the data pipeline, you will have a count. Earlier, 162 records passed through the data pipeline when I ran it. To see your counters, click the waffle menu in the top-right corner of the NiFi window and select **Counters**, as shown in the following screenshot:

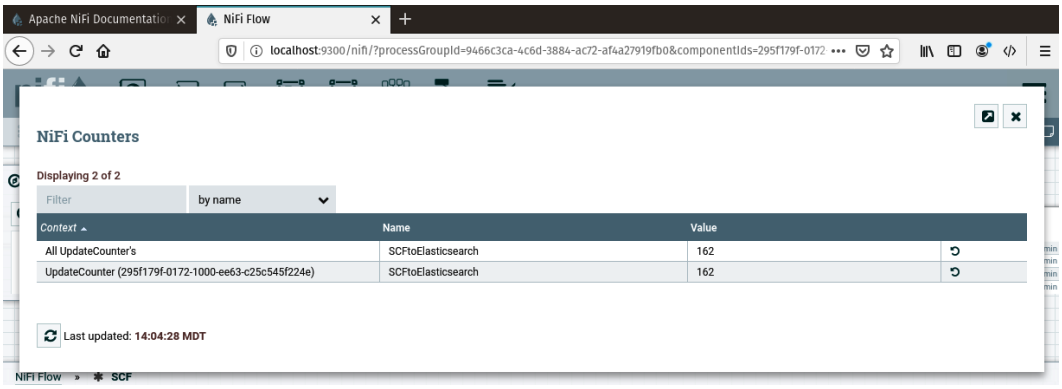

Figure 9.10 – NiFi Counters

The preceding screenshot shows the counts of the counter and an aggregate. If we had other `UpdateCounter` processors that updated the same counter, it would aggregate those values.

Using the GUI is an excellent way to quickly see how your processor groups and processors are running. But you can also use processors to monitor the data pipeline.

In the previous section, you learned about the NiFi bulletin. You can use background tasks to monitor NiFi and post that data to the NiFi bulletin using reporting tasks. Reporting tasks are like processors that run in the background and perform a task. The results will appear in the bulletin or you can send it to other locations.

To create a reporting task, in the waffle menu, select **Controller Settings**, then navigate to the **Reporting Task** tab. The list should be blank, but you can add a new task using the plus sign on the right-hand corner of the window. You will see a list of tasks that are available. Single-click on one to see the description. You should see a list similar to the following screenshot:

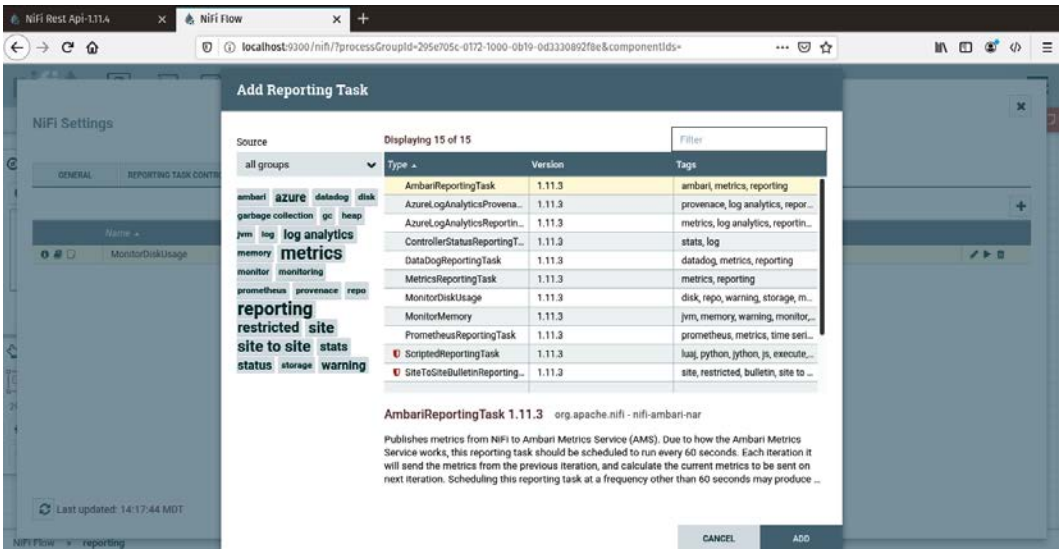

Figure 9.11 – Reporting tasks available in NiFi

For this example, double click the **MonitorDiskUsage** task. It will appear on the list with the ability to edit. Click the pencil to edit, set the **Threshold** to **1%**, and set the directory to your NiFi directory. The configuration will look like the following screenshot:

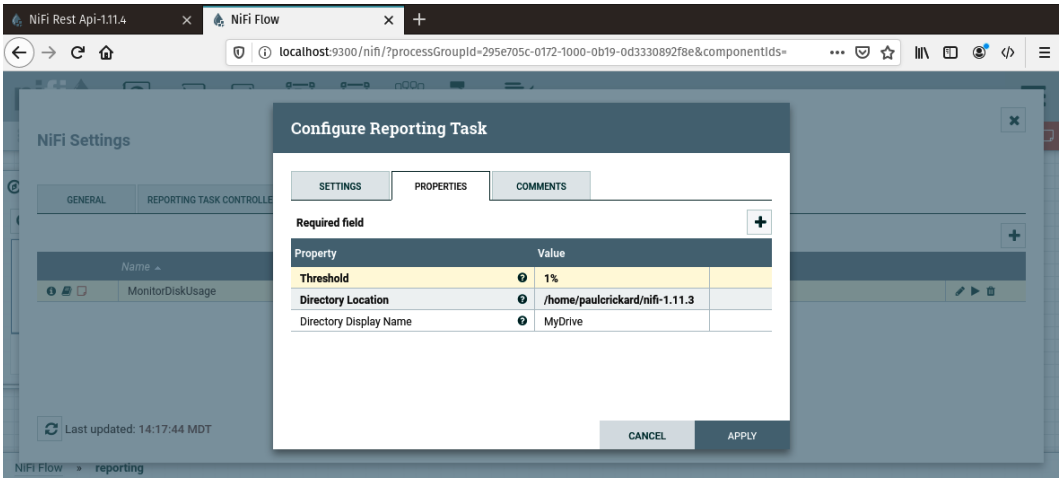

Figure 9.12 – Configuring the MonitorDiskUsage task

You can use a percentage or a value such as 20 gigabytes. I have set it to 1% so that it will post to the bulletin. I chose the NiFi directory because it contains all the logs and repositories.

Running the **Reporting Task** processor, you should see a bulletin in the main NiFi window. The message will be that the **MonitorDiskUsage** task exceeded the 1% threshold. The following screenshot shows the bulletin:

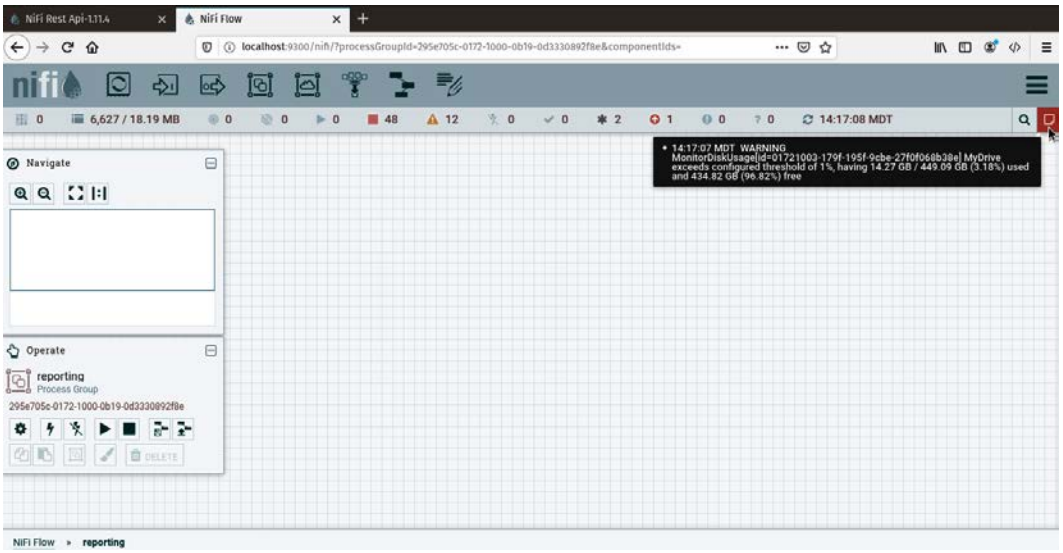

Figure 9.13 – The Reporting Task bulletin

You can create reporting tasks for many other metrics. Using the GUI is useful and convenient, but you will most likely not be able to sit in front of NiFi watching it all day. This would be horribly inefficient. A better method would be to have NiFi send you a message. You can do this using processors. The next section will show you how.

Monitoring NiFi with processors

Instead of relying on watching the NiFi GUI, you can insert a processor into your data pipeline to report what is happening with the pipeline. For example, you can use the `PutSlack` processor to send messages on failures or success.

To send Slack messages, you will need to create an app in your Slack workspace. You can do this by browsing to `https://api.slack.com/apps`. Click **Create New App**, as shown in the following screenshot:

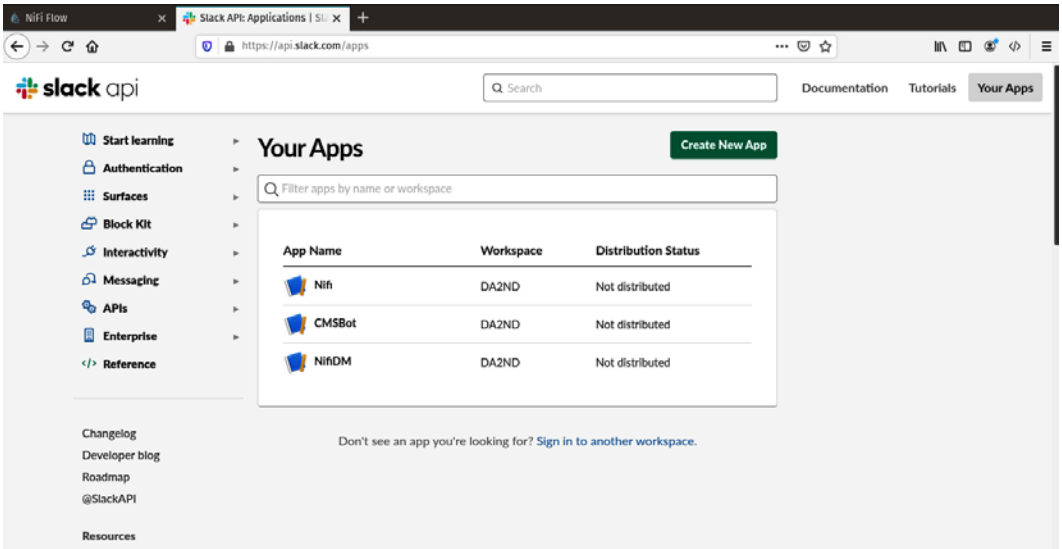

Figure 9.14 – Creating a new app

Slack will ask you to name your app and then select a workspace, as shown in the following screenshot:

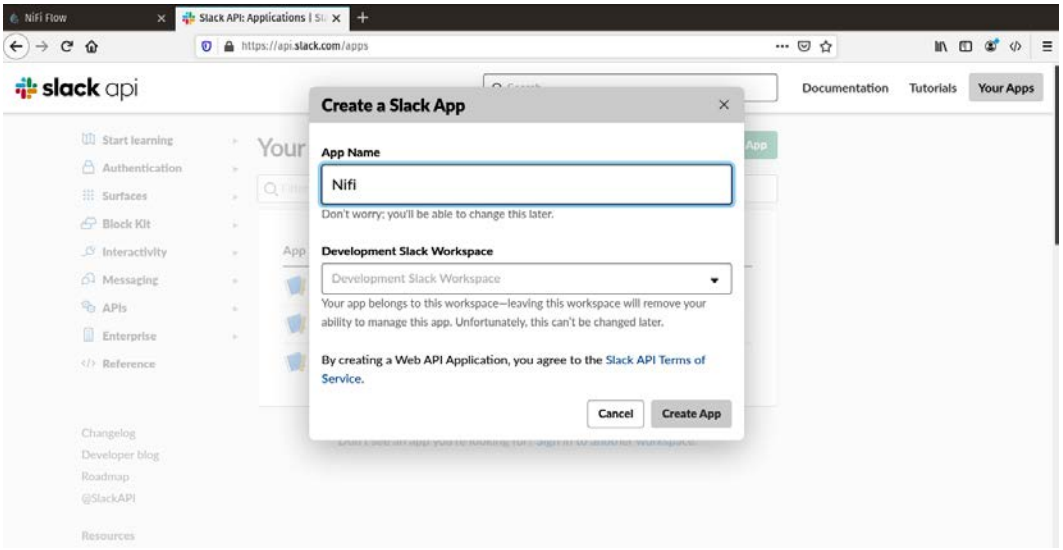

Figure 9.15 – Specifying a name and workspace for your app

When finished, you will be redirected to the app page. Under the **Features** heading, click **Incoming Webhooks** and turn it on, as shown in the following screenshot:

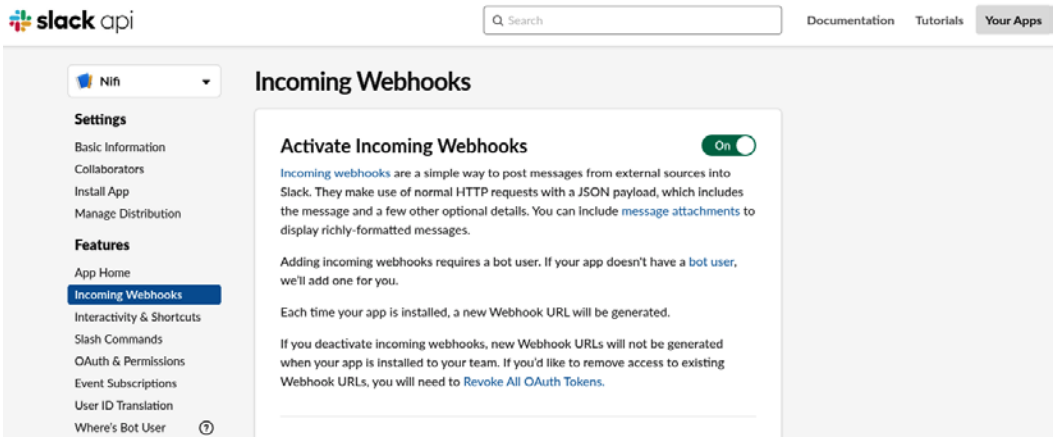

Figure 9.16 – Activating Incoming Webhooks

You will be asked to select a channel for the webhook. I selected myself so that the channel becomes a direct message to me. You can create a channel for the data pipeline so that multiple people can join and see the messages. Once you have completed this step, scroll to the bottom of the page to see the new webhook. Click the copy button and open NiFi. It is time to add `PutSlack` to your data pipeline.

In NiFi, I have opened the **SCF** processor group. I found the **ElasticSCF** processor—the processor that sends the issues in to Elasticsearch. Drag and drop the processor icon in the control toolbar to the canvas and select **PutSlack**. Create a connection between **ElasticSCF** and **PutSlack** for the relationship failure, as shown in the following screenshot:

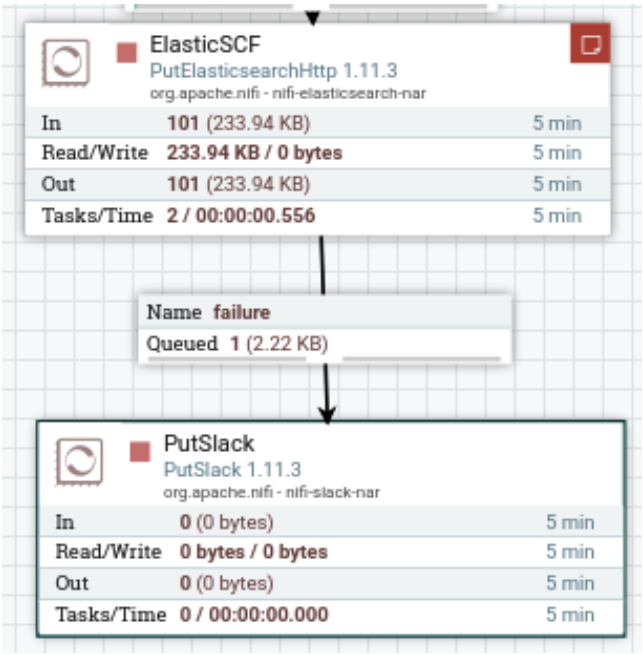

Figure 9.17 – PutSlack added to the end of the data pipeline

To configure the `PutSlack` processor, paste the copied URL to the **Webhook URL** property. NiFi will hide the URL because it is a sensitive property. The **Username** property is whatever you want Slack to display when the message is sent. You can also set an icon or an emoji. The **Webhook Text** property is the message that will be sent. You can set the message to plain text saying that the processor failed, or because the **Webhook Text** property accepts the NiFi expression language, you can use a combination of a flowfile attribute with text. I have configured the processor as shown in the following screenshot:

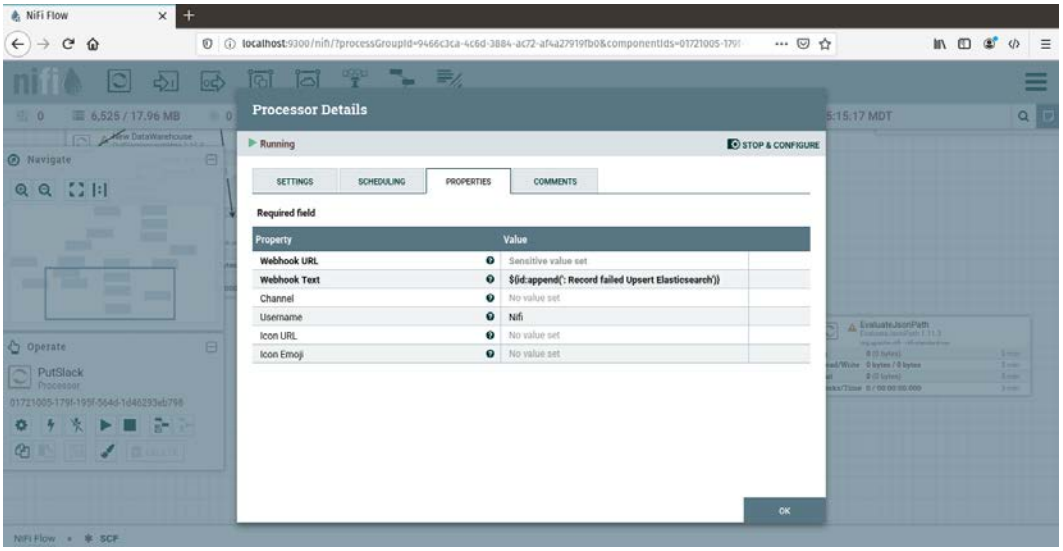

Figure 9.18 – PutSlack configuration

I used the append method of the NiFi expression language. The statement is as follows:

```
${id:append(': Record failed Upsert Elasticsearch')}
```

The preceding statement gets the `id` property, `${id}`, and calls append, `:append()`. Inside the `append()` method is the text. The result will be a message like the one shown in the following screenshot:

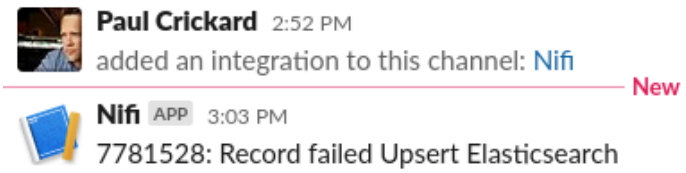

Figure 9.19 – Slack direct message from NiFi

The preceding screenshot is my direct messages. You can see that I added the NiFi integration to the workspace, then received a message from NiFi. The message is the ID of the **SeeClickFix** issue and some text saying that it failed. I can now take action.

You can use processors to send emails, write files, or perform many other actions that you could use to monitor your data pipeline. You can also write your own monitoring applications outside of NiFi using Python. The next section will cover the NiFi REST API.

Using Python with the NiFi REST API

Using Python and the NiFi REST API, you could write your own monitoring tools, or wire up a dashboard. The NiFi REST API documentation is located at `https://nifi.apache.org/docs/nifi-docs/rest-api/index.html`. You can see all of the different endpoints by type and some information about each of them. This section will highlight some of the endpoints that you have covered in this chapter but by using the GUI.

The first thing we can look at are the system diagnostics. System diagnostics will show you your resource usage. You can see heap size, threads, repository usage, and several other metrics. To call the endpoint with requests, you can use the following code:

```
r=requests.get('http://localhost:9300/nifi-api/system-
diagnostics')
data=r.json()
data['systemDiagnostics']['aggregateSnapshot']['maxHeap']
#'512 MB'
data['systemDiagnostics']['aggregateSnapshot']['totalThreads']
#108
data['systemDiagnostics']['aggregateSnapshot']
['heapUtilization']
#'81.0%'
```

Other endpoints of interest are the processor groups. Using this endpoint, you can find basic information about any processor group. You will need to get the ID of the group in NiFi. You can find this in the URL as the `processGroupId` parameter. With it, you can use the process-groups endpoint, as shown in the following code:

```
pg=requests.get('http://localhost:9300/nifi-api/process-
groups/9466c3ca-4c6d-3884-ac72-af4a27919fb0')
pgdata=pg.json()
pgdata['component']['name']
#'SCF'
pgdata['status']
```

The `status` object holds most of the pertinent information that you would find in the status toolbar. The output is as follows:

```
{'id': '9466c3ca-4c6d-3884-ac72-af4a27919fb0', 'name': 'SCF',
'statsLastRefreshed': '16:11:16 MDT', 'aggregateSnapshot':
{'id': '9466c3ca-4c6d-3884-ac72-af4a27919fb0', 'name': 'SCF',
```

```
'versionedFlowState': 'LOCALLY_MODIFIED', 'flowFilesIn':
0, 'bytesIn': 0, 'input': '0 (0 bytes)', 'flowFilesQueued':
6481, 'bytesQueued': 18809602, 'queued': '6,481 (17.94 MB)',
'queuedCount': '6,481', 'queuedSize': '17.94 MB', 'bytesRead':
0, 'read': '0 bytes', 'bytesWritten': 0, 'written': '0
bytes', 'flowFilesOut': 0, 'bytesOut': 0, 'output': '0 (0
bytes)', 'flowFilesTransferred': 0, 'bytesTransferred':
0, 'transferred': '0 (0 bytes)', 'bytesReceived': 0,
'flowFilesReceived': 0, 'received': '0 (0 bytes)',
'bytesSent': 0, 'flowFilesSent': 0, 'sent': '0 (0 bytes)',
'activeThreadCount': 0, 'terminatedThreadCount': 0}}
```

Using the processors endpoint, you can look specifically at a single processor. You can use the `status` object to look at the status toolbar information, as shown in the following code:

```
p=requests.get('http://localhost:9300/nifi-api/
processors/8b63e4d0-eff2-3093-f4ad-0f1581e56674')
pdata=p.json()
pdata['component']['name']
#'Query SCF - Archive'
pdata['status']
```

Using the NiFi API, you can even look into the queues and download flowfiles. To get the contents of a flowfile you need to follow these steps:

1. Make a listing request to the queue:

```
q=requests.post('http://localhost:9300/nifi-api/flowfile-
queues/295fc119-0172-1000-3949-54311cdb478e/listing-
requests')
qdata=q.json()
listid=qdata['listingRequest']['id']
# '0172100b-179f-195f-b95c-63ea96d151a3'
```

2. Then you will get the status of the listing request by passing the request (`listid`):

```
url="http://localhost:9300/nifi-api/flowfile-
queues/295fc119-0172-1000-3949-54311cdb478e/listing-
requests/"+listid
ff=requests.get(url)
ffdata=ff.json()
ffid=ffdata['listingRequest']['flowFileSummaries'][0]
```

```
['uuid']
#'3b2dd0fa-dfbe-458b-83e9-ea5f9dbb578f'
```

3. Lastly, you will call the flowfiles endpoint, pass the flowfile ID (`ffid`), and then request the content. The flowfile is JSON, so the result will be JSON:

```
ffurl="http://localhost:9300/nifi-api/flowfile-
queues/295fc119-0172-1000-3949-54311cdb478e/
flowfiles/"+ffid+"/content"
download=requests.get(ffurl)
download.json()
```

You now have the contents of an entire flowfile:

```
{'request_type': {'related_issues_url': 'https://
seeclickfix.com/api/v2/issues?lat=35.18151754051&lng=-
106.689667822892&request_types=17877&sort=distance',
'title': 'Missed Trash Pick Up', 'url': 'https://
seeclickfix.com/api/v2/request_types/17877',
'organization': 'City of Albuquerque', 'id': 17877},
'shortened_url': None, 'rating': 2, 'description': 'Yard
waste in bags', 'created_at': '2020-05-08T17:15:57-
04:00', 'opendate': '2020-05-08', 'media': {'image_
square_100x100': None, 'image_full': None, 'video_
url': None, 'representative_image_url': 'https://
seeclickfix.com/assets/categories/trash-f6b4bb46a3084
21d38fc042b1a74691fe7778de981d59493fa89297f6caa86a1.
png'}, 'private_visibility': False, 'transitions':
{}, 'point': {'coordinates': [-106.689667822892,
35.18151754051], 'type': 'Point'}, 'updated_at':
'2020-05-10T16:31:42-04:00', 'id': 7781316, 'lat':
35.18151754051, 'coords': '35.1815175405,-106.689667823',
'summary': 'Missed Trash Pick Up', 'address': '8609 Tia
Christina Dr Nw Albuquerque NM 87114, United States',
'closed_at': '2020-05-08T17:24:55-04:00', 'lng':
-106.689667822892, 'comment_url': 'https://seeclickfix.
com/api/v2/issues/7781316/comments', 'reporter':
{'role': 'Registered User', 'civic_points': 0, 'avatar':
{'square_100x100': 'https://seeclickfix.com/assets/
no-avatar-100-5e06fcc664c6376bbf654cbd67df857ff81918c5f5c
6a2345226093147382de9.png', 'full': 'https://seeclickfix.
com/assets/no-avatar-100-5e06fcc664c6376bbf654cbd67d
f857ff81918c5f5c6a2345226093147382de9.png'}, 'html_
url': 'https://seeclickfix.com/users/347174', 'name':
'Gmom', 'id': 347174, 'witty_title': ''}, 'flag_url':
```

```
'https://seeclickfix.com/api/v2/issues/7781316/flag',
'url': 'https://seeclickfix.com/api/v2/issues/7781316',
'html_url': 'https://seeclickfix.com/issues/7781316',
'acknowledged_at': '2020-05-08T17:15:58-04:00', 'status':
'Archived', 'reopened_at': None}
```

4. You can clear queues by making a `drop` request:

```
e=requests.post('http://localhost:9300/nifi-api/flowfile-
queues/295fc119-0172-1000-3949-54311cdb478e/drop-
requests')
edata=e.json()
```

5. You can pass the listing request ID to the end of the preceding URL to see that it worked. Or you can open NiFi and browse to the queue and you will see that it is empty.
6. You can read the NiFi bulletin by calling the bulletin board endpoint:

```
b=requests.get('http://localhost:9300/nifi-api/flow/
bulletin-board')
bdata=b.json()
bdata
```

The result is a single message saying I do not have Elasticsearch running. The output is as follows:

```
{'bulletinBoard': {'bulletins': [{'id': 2520, 'groupId':
'9466c3ca-4c6d-3884-ac72-af4a27919fb0', 'sourceId':
'e5fb7c4b-0171-1000-ac53-9fd365943393', 'timestamp':
'17:15:44 MDT', 'canRead': True, 'bulletin': {'id': 2520,
'category': 'Log Message', 'groupId': '9466c3ca-4c6d-
3884-ac72-af4a27919fb0', 'sourceId': 'e5fb7c4b-0171-1000-
ac53-9fd365943393', 'sourceName': 'ElasticSCF', 'level':
'ERROR', 'message': 'PutElasticsearchHttp[id=e5fb7c4b-
0171-1000-ac53-9fd365943393] Routing to failure due to
exception: Failed to connect to localhost/127.0.0.1:9200:
java.net.ConnectException: Failed to connect to
localhost/127.0.0.1:9200', 'timestamp': '17:15:44
MDT'}}], 'generated': '17:16:20 MDT'}}
```

7. You can also read the counters you created earlier. The following code will send a `get` request to the counter endpoint:

```
c=requests.get('http://localhost:9300/nifi-api/counters')
cdata=c.json()
cdata
```

In the following code block, you will see that I have added an additional counter:

```
{'counters': {'aggregateSnapshot': {'generated':
'17:17:17 MDT', 'counters': [{'id': '6b2fdf54-a984-
38aa-8c56-7aa4a544e8a3', 'context': 'UpdateCounter
(01721000-179f-195f-6715-135d1d999e33)', 'name':
'SCFSplit', 'valueCount': 1173, 'value': '1,173'}, {'id':
'b9884362-c70e-3634-8e53-f0151396be0b', 'context': "All
UpdateCounter's", 'name': 'SCFSplit', 'valueCount':
1173, 'value': '1,173'}, {'id': 'fb06d19f-682c-3f85-9ea2-
f12b090c4abd', 'context': "All UpdateCounter's", 'name':
'SCFtoElasticsearch', 'valueCount': 162, 'value': '162'},
{'id': '72790bbc-3115-300d-947c-22d889f15a73', 'context':
'UpdateCounter (295f179f-0172-1000-ee63-c25c545f224e)',
'name': 'SCFtoElasticsearch', 'valueCount': 162, 'value':
'162'}]}}}
```

8. Lastly, you can also get information on your reporting tasks. You can see the results in the bulletin, but this endpoint allows you to see their state; in this case, I have them stopped. The following code shows you how:

```
rp=requests.get('http://localhost:9300/nifi-api/
reporting-tasks/01721003-179f-195f-9cbe-27f0f068b38e')
rpdata=rp.json()
rpdata
```

The information about the reporting task is as follows:

```
{'revision': {'clientId': '2924cbec-0172-1000-ab26-
103c63d8f745', 'version': 8}, 'id': '01721003-179f-195f-
9cbe-27f0f068b38e', 'uri': 'http://localhost:9300/
nifi-api/reporting-tasks/01721003-179f-195f-9cbe-
27f0f068b38e', 'permissions': {'canRead': True,
'canWrite': True}, 'bulletins': [], 'component':
{'id': '01721003-179f-195f-9cbe-27f0f068b38e',
'name': 'MonitorDiskUsage', 'type': 'org.apache.nifi.
controller.MonitorDiskUsage', 'bundle': {'group':
'org.apache.nifi', 'artifact': 'nifi-standard-nar',
```

```
'version': '1.12.1'}, 'state': 'STOPPED', 'comments':
'', 'persistsState': False, 'restricted': False,
'deprecated': False, 'multipleVersionsAvailable': False,
'schedulingPeriod': '5 mins', 'schedulingStrategy':
'TIMER_DRIVEN', 'defaultSchedulingPeriod': {'TIMER_
DRIVEN': '0 sec', 'CRON_DRIVEN': '* * * * * ?'},
'properties': {'Threshold': '1%', 'Directory Location':
'/home/paulcrickard/nifi-1.12.1', 'Directory Display
Name': 'MyDrive'}, 'descriptors': {'Threshold': {'name':
'Threshold', 'displayName': 'Threshold', 'description':
'The threshold at which a bulletin will be generated to
indicate that the disk usage of the partition on which
the directory found is of concern', 'defaultValue':
'80%', 'required': True, 'sensitive': False, 'dynamic':
False, 'supportsEl': False, 'expressionLanguageScope':
'Not Supported'}, 'Directory Location': {'name':
'Directory Location', 'displayName': 'Directory
Location', 'description': 'The directory path of
the partition to be monitored.', 'required': True,
'sensitive': False, 'dynamic': False, 'supportsEl':
False, 'expressionLanguageScope': 'Not Supported'},
'Directory Display Name': {'name': 'Directory Display
Name', 'displayName': 'Directory Display Name',
'description': 'The name to display for the directory in
alerts.', 'defaultValue': 'Un-Named', 'required': False,
'sensitive': False, 'dynamic': False, 'supportsEl':
False, 'expressionLanguageScope': 'Not Supported'}},
'validationStatus': 'VALID', 'activeThreadCount': 0,
'extensionMissing': False}, 'operatePermissions':
{'canRead': True, 'canWrite': True}, 'status':
{'runStatus': 'STOPPED', 'validationStatus': 'VALID',
'activeThreadCount': 0}}
```

With these NiFi endpoints, you can collect information on your system, on process groups, on processors, and on queues. You can use this information to build your own monitoring systems or create dashboards. The API has a lot of potential—you could even call the API using NiFi itself.

Summary

In this chapter, you have learned how to use the NiFi GUI to monitor your data pipelines using the status bar, the bulletin, and counters. You also learned how to add processors that can send information to you inside your data pipeline. With the `PutSlack` processor, you were able to send yourself direct messages when there was a failure, and you passed data from the flowfile in the message with the NiFi expression language. Lastly, you learned how to use the API to write your own monitoring tools and grab the same data as is in the NiFi GUI—even reading the contents of a single flowfile.

In the next chapter, you will learn how to deploy your production pipelines. You will learn how to use processor groups, templates, versions, and variables to allow you to import data pipelines to a production NiFi instance with minimal configuration.

10
Deploying Data Pipelines

In software engineering, you will usually have **development**, **testing**, and **production** environments. The testing environment may be called **quality control** or **staging** or some other name, but the idea is the same. You develop in an environment, then push it to another environment that will be a clone of the production environment and if everything goes well, then it is pushed into the production environment. The same methodology is used in data engineering. So far, you have built data pipelines and run them on a single machine. In this chapter, you will learn methods for building data pipelines that can be deployed to a production environment.

In this chapter, we're going to cover the following main topics:

- Finalizing your data pipelines for production
- Using the NiFi variable registry
- Deploying your data pipelines

Finalizing your data pipelines for production

In the last few chapters, you have learned about the features and methods for creating production data pipelines. There are still a few more features needed before you can deploy your data pipelines—backpressure, processor groups with input and output ports, and funnels. This section will walk you through each one of these features.

Backpressure

In your data pipelines, each processor or task will take different amounts of time to finish. For example, a database query may return hundreds of thousands of results that are split into single flowfiles in a few seconds, but the processor that evaluates and modifies the attributes within the flowfiles may take much longer. It doesn't make sense to dump all of the data into the queue faster than the downstream processor can actually process it. Apache NiFi allows you to control the number of flowfiles or the size of the data that is sent to the queue. This is called **backpressure**.

To understand how backpressure works, let's make a data pipeline that generates data and writes it to a file. The data pipeline is shown in the following screenshot:

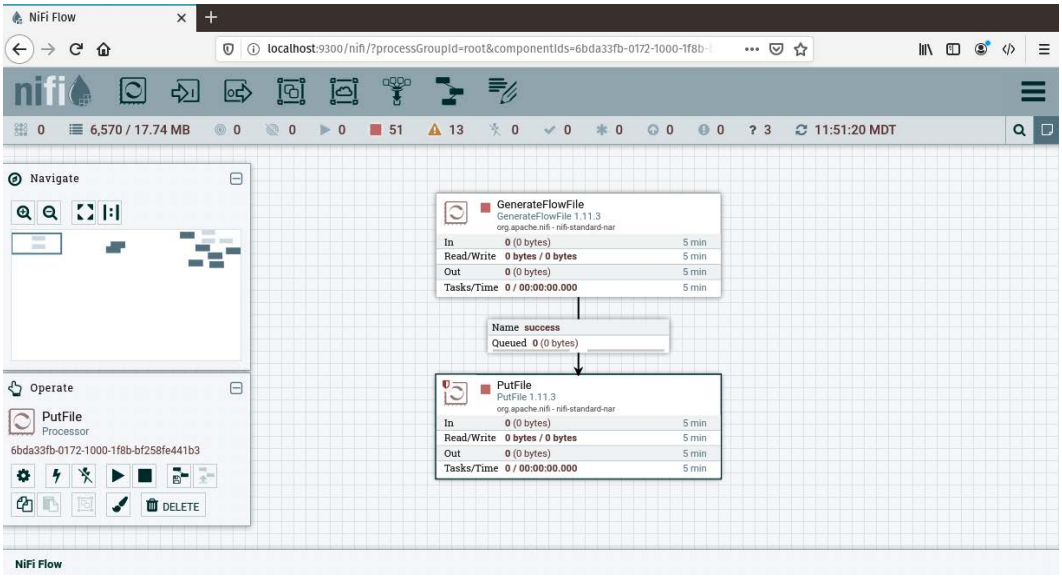

Figure 10.1 – A data pipeline to generate data and write the flowfiles to a file

The preceding data pipeline a creates connection between the `GenerateFlowFile` processor and the `PutFile` processor for the success relationship. I have configured the `PutFile` processor to write files to `/home/paulcrickard/output`. The `GenerateFlowFile` processor is using the default configuration.

If you run the data pipeline by starting the `GenerateFlowFile` processor only, you will see that the queue has 10,000 flowfiles and is red, as shown in the following screenshot:

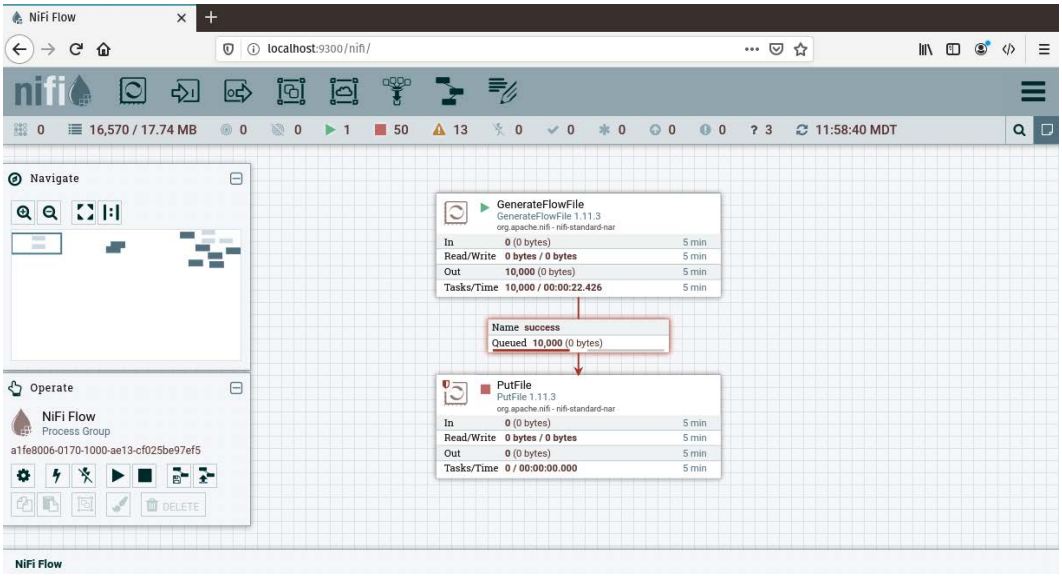

Figure 10.2 – A full queue with 10,000 flowfiles

If you refresh NiFi, the number of flowfiles in the queue will not increase. It has 10,000 flowfiles and cannot hold anymore. But is 10,000 the maximum number?

Queues can be configured just like the processors that feed them. Right-click on the queue and select **Configure**. Select the **SETTINGS** tab, and you will see the following options:

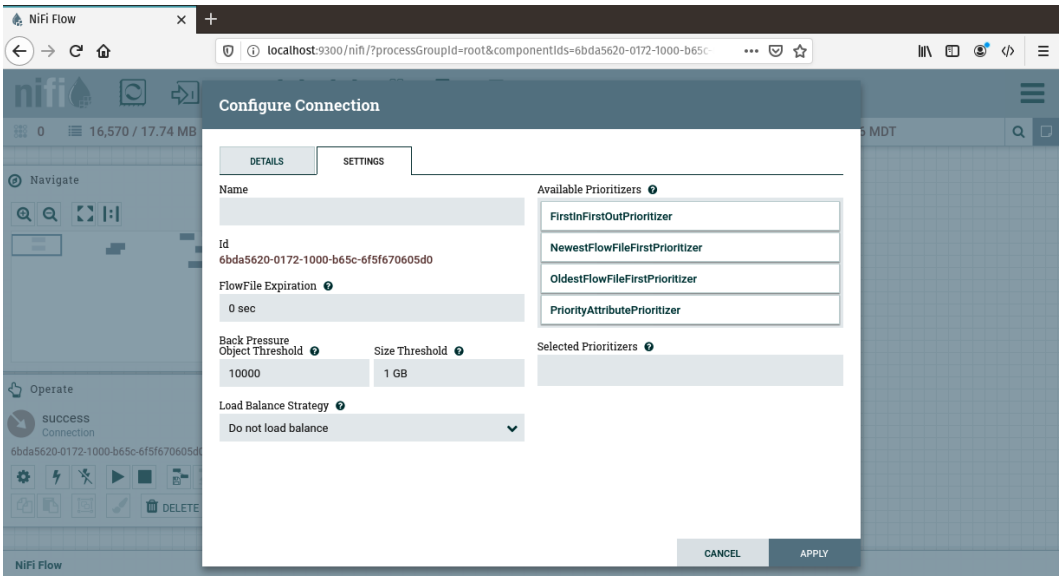

Figure 10.3 – Queue configuration settings

You will notice that **Back Pressure Object Threshold** is set to `10000` flowfiles and that **Size Threshold** is set to `1 GB`. The `GenerateFlowFile` processor set the size of each flowfile to 0 bytes, so the object threshold was hit before the size threshold. You can test hitting the size threshold by changing the **File Size** property in the `GenerateFlowFile` processor. I have changed it to 50 MB. When I start the processor, the queue now stops at 21 flowfiles because it has exceeded 1 GB of data. The following screenshot shows the full queue:

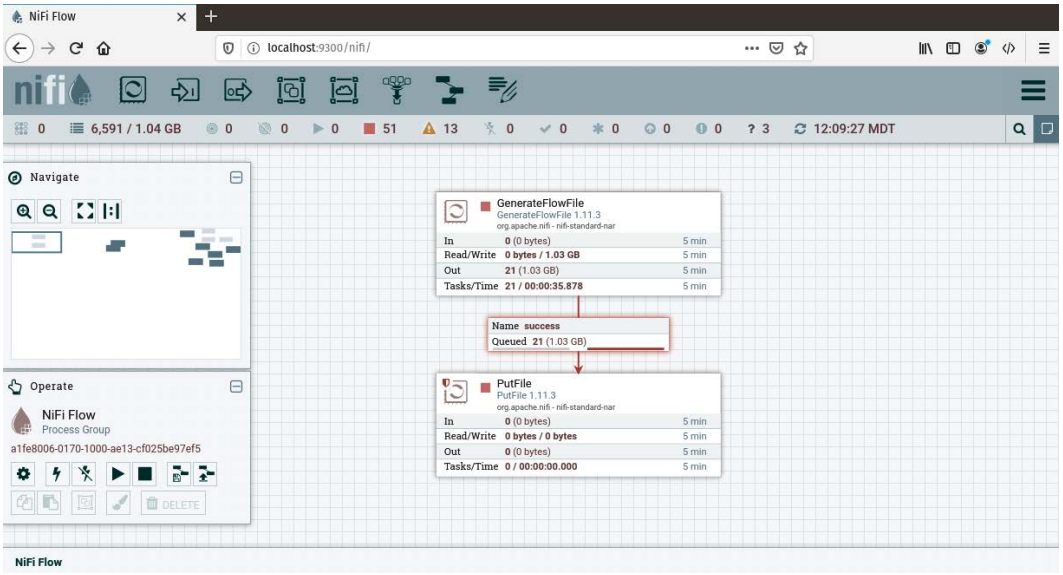

Figure 10.4 – Queue that has the size threshold

By adjusting **Object Threshold** or **Size Threshold**, you can control the amount of data that gets sent to a queue and create backpressure slowing down an upstream processor. While loading the queues does not break your data pipeline, it will run much more smoothly if the data flows in a more even manner.

The next section will zoom out on your data pipelines and show other techniques for improving the use of processor groups.

Improving processor groups

Up to this point, you have used processor groups to hold a single data pipeline. If you were to push all of these data pipelines to production, what you would soon realize is that you have a lot of processors in each processor group doing the same exact task. For example, you may have several processors that SplitJson used followed by an `EvaluateJsonPath` processor that extracts the ID from a flowfile. Or, you might have several processors that insert flowfiles in to Elasticsearch.

You would not have several functions in code that do the exact same thing on different variables; you would have one that accepted parameters. The same holds true for data pipelines, and you accomplish this using processor groups with the input and output ports.

To illustrate how to break data pipelines into logical pieces, let's walk through an example:

1. In NiFi, create a processor group and name it `Generate Data`.
2. Inside the processor group, drag the `GenerateFlowFile` processor to the canvas. I have set the **Custom Text** property in the configuration to `{"ID":123}`.
3. Next, drag an output port to the canvas. You will be prompted for **Output Port Name** and **Send To**. I have named it `FromGeneratedData` and **Send To** is set to **Local connections**.
4. Lastly, connect the `GenerateFlowfile` processor to **Output Port**. You will have a warning on the output port that it is invalid because it has no outgoing connections. We will fix that in the next steps.
5. Exit the processor group.
6. Create a new processor group and name it `Write Data`.
7. Enter the processor group and drag the `EvaluateJsonPath` processor to the canvas. Configure it by creating a property ID with the value of `$.{ID}`, and set the **Destination** property to **flowfile-attribute**.
8. Next, drag the `UpdateAttribute` processor to the canvas and create a new property filename and set the value to `${ID}`.
9. Now, drag the `PutFile` processor to the canvas. Set the **Directory** property to any location you have permissions to edit. I have set mine to `/home/paulcrickard/output`.

10. Lastly, drag an **Input Port** to the canvas and make it the first processor in the data pipeline. The completed pipeline should look like the following screenshot:

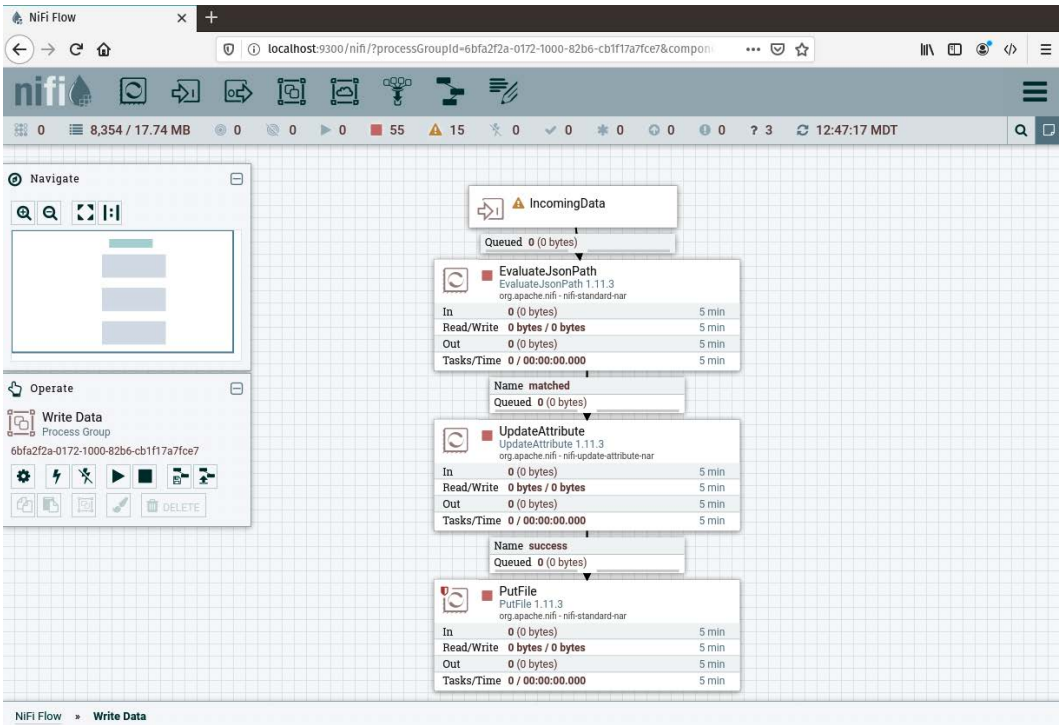

Figure 10.5 – A data pipeline that starts with an input port

11. Exit the processor group. You should now have two processor groups on the canvas—`Generate Data` and `Write Data`. You can connect these processor groups just like you do with single processors. When you connect them by dragging the arrow from `Generate Data` to `Write Data`, you will be prompted to select which ports to connect, as shown in the following screenshot:

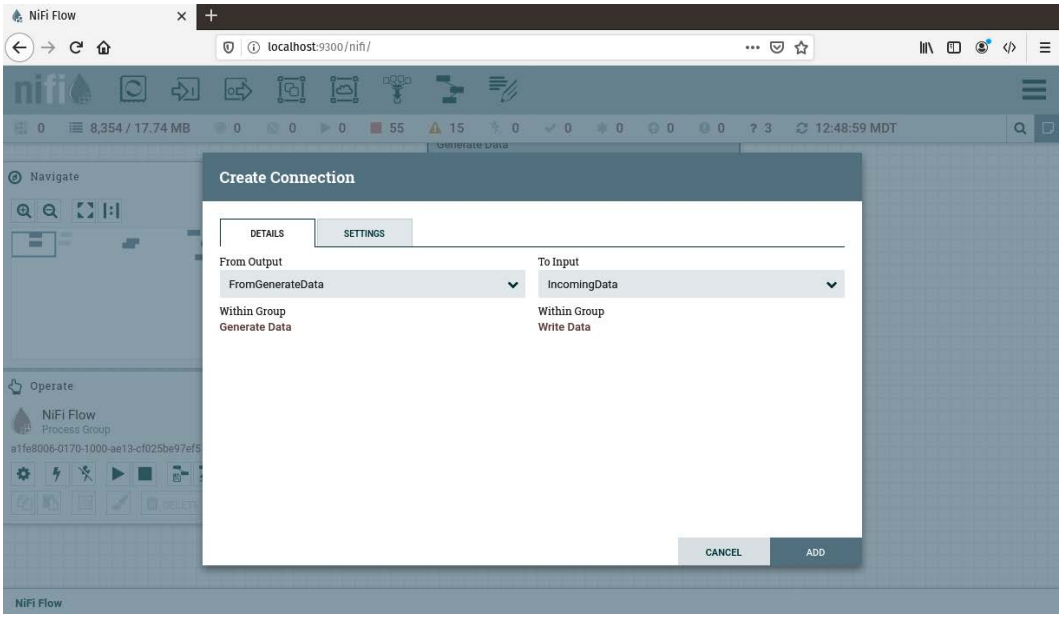

Figure 10.6 – Connecting two processor groups

12. The default values will work because you only have one output port and one input port. If you had more, you could use the drop-down menus to select the proper ports. This is where naming them something besides input and output becomes important. Make the names descriptive.
13. With the processor groups connected, start the `Generate Data` group only. You will see the queue fill up with flowfiles. To see how the ports work, enter the `Write Data` processor group.
14. Start only the incoming data input port. Once it starts running, the downstream queue will fill with flowfiles.

15. Right-click the queue and select **List queue**. You can see that the flowfiles are coming from the `Generate Data` processor group. You can now start the rest of the processor.
16. As the data pipeline runs, you will have a file, `123`, created in your output directory.

You have successfully connected two processor groups using input and output ports. In production, you can now have a single process group to write data to a file and it can receive data from any processor group that needs to write data, as shown in the following screenshot:

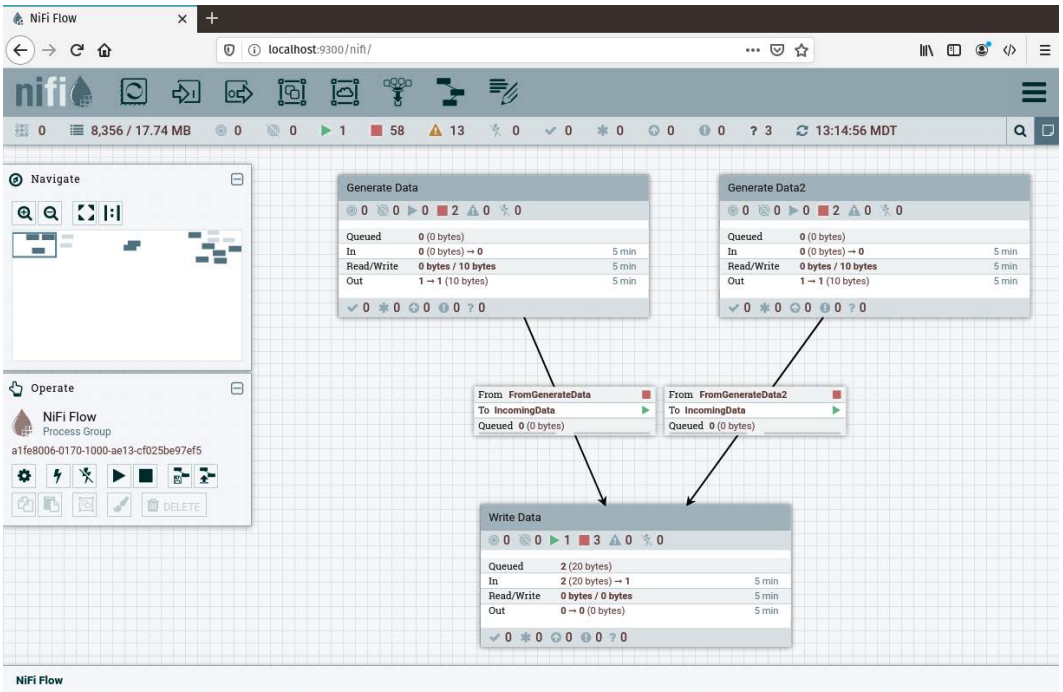

Figure 10.7 – Two processor groups utilizing the Write Data processor group

In the preceding data pipeline, I made a copy of `Generate Data` and configured the **Custom Text** property to `{"ID":456}` and set the run schedule to an hour so that I would only get one flowfile from each processor—`Generate Data` and `Generate Data2`. Running all of the processor groups, you list the queue and confirm that one flowfile comes from each processor group, and your output directory now has two files—`123` and `456`.

Using the NiFi variable registry

When you are building your data pipelines, you are hardcoding variables—with the exception of some expression language where you extract data from the flowfile. When you move the data pipeline to production, you will need to change the variables in your data pipeline, and this can be time consuming and error prone. For example, you will have a different test database than production. When you deploy your data pipeline to production, you need to point to production and change the processor. Or you can use the variable registry.

Using the `postgresToelasticsearch` processor group from *Chapter 4, Working with Databases*, I will modify the data pipeline to use the NiFi variable registry. As a reminder, the data pipeline is shown in the following screenshot:

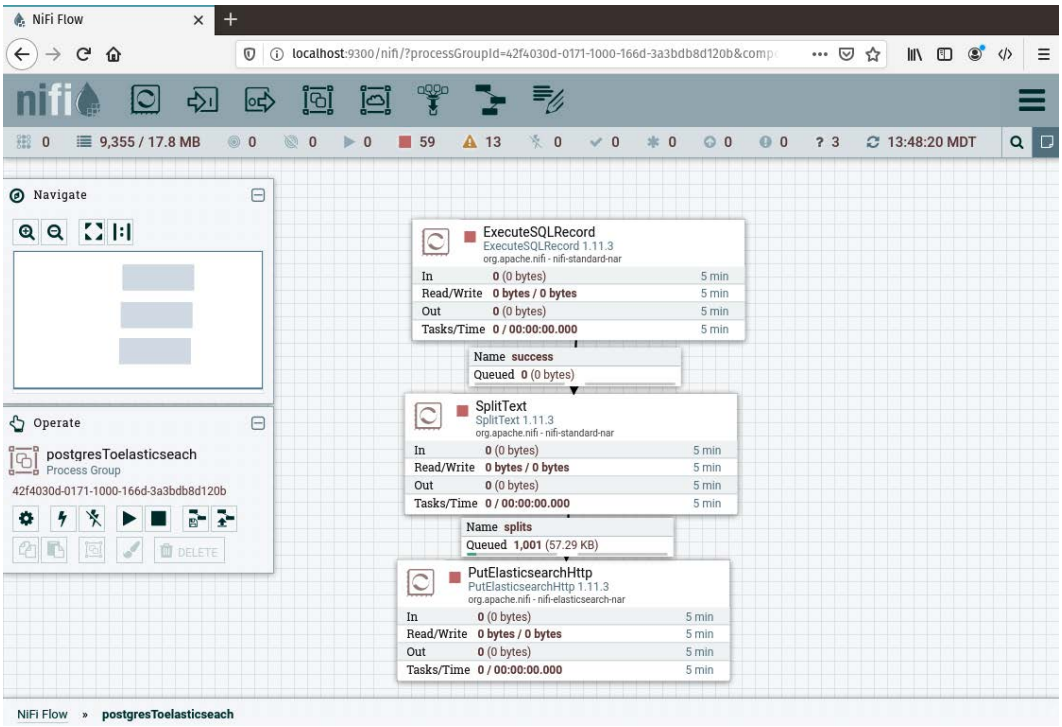

Figure 10.8 – A data pipeline to query PostgreSQL and save the results to Elasticsearch

From outside the processor group, right-click on it and select **Variables**. To add a new variable, you can click the plus sign and provide a name and a value. These variables are now associated with the processor group.

Just like functions in programming, variables have a scope. Variables in a processor group are local variables. You can right-click on the NiFi canvas and create a variable, which you can consider global in scope. I have created two local variables, `elastic` and `index`, and one global, `elastic`. When I open the variables in the group, it looks like the following screenshot:

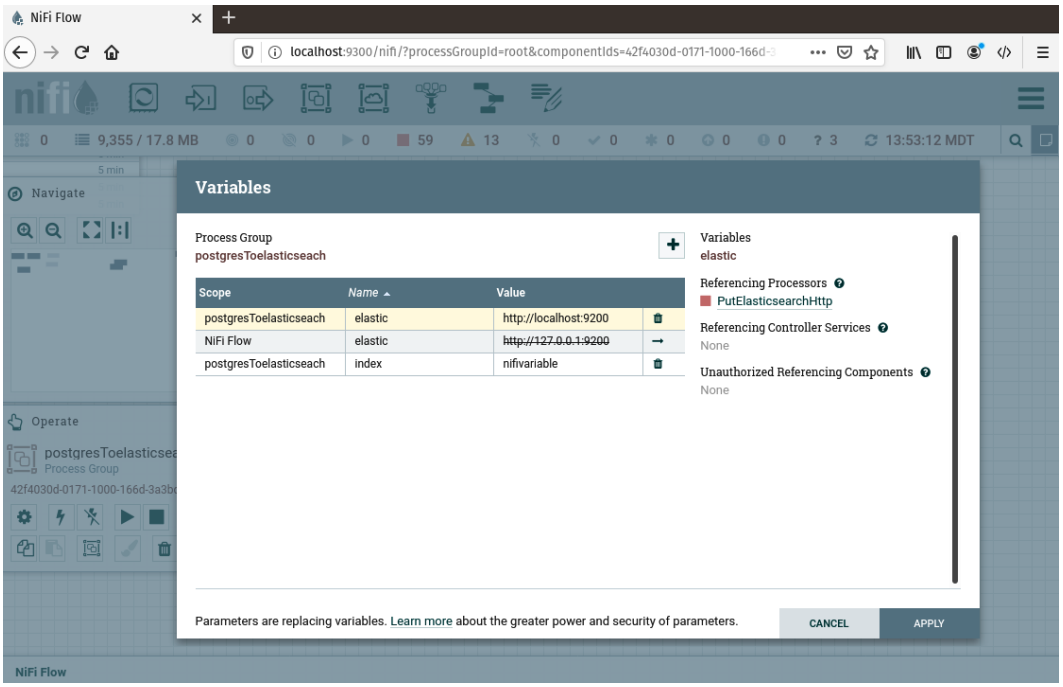

Figure 10.9 – NiFi variable registry

In the preceding screenshot, you can see the scopes. The scope of **postgresToelasticsearch** is the processor group, or local variables. The **NiFi Flow** scope is the global variables. Because I have two variables named `elastic`, the local variable takes precedence.

You can now reference these variables using the expression language. In the `PutElasticsearchHttp` process, I have set the **Elasticsearch URL** to `${elastic}` and the **Index** to `${index}`. These will populate with the local variables—`http://localhost:9200` and `nifivariable`.

Running the data pipeline, you can see the results in Elasticsearch. There is now a new index with the name `nifivariable` and 1,001 records. The following screenshot shows the result:

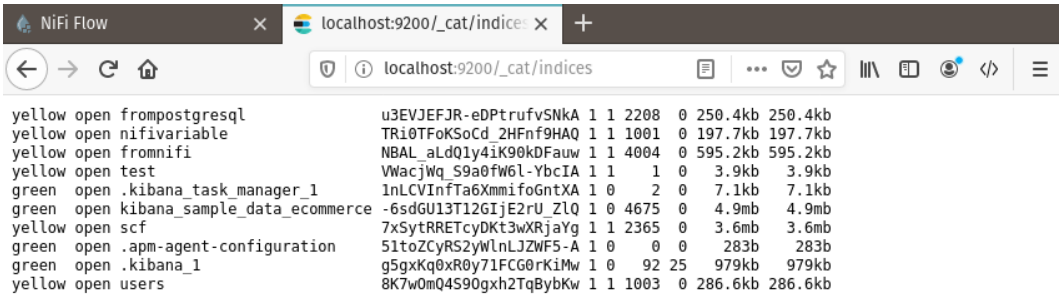

Figure 10.10 – The new index, nifivariable, is the second row

You have now put the finishing touches on production pipelines and have completed all the steps needed to deploy them. The next section will teach you different ways to deploy your data pipelines.

Deploying your data pipelines

There are many ways to handle the different environments—**development**, **testing**, **production**—and how you choose to do that is up to what works best with your business practices. Having said that, any strategy you take should involve using the NiFi registry.

Using the simplest strategy

The simplest strategy would be to run NiFi over the network and split the canvas into multiple environments. When you have promoted a process group, you would move it in to the next environment. When you needed to rebuild a data pipeline, you would add it back to development and modify it, then update the production data pipeline to the newest version. Your NiFi instance would look like the following screenshot:

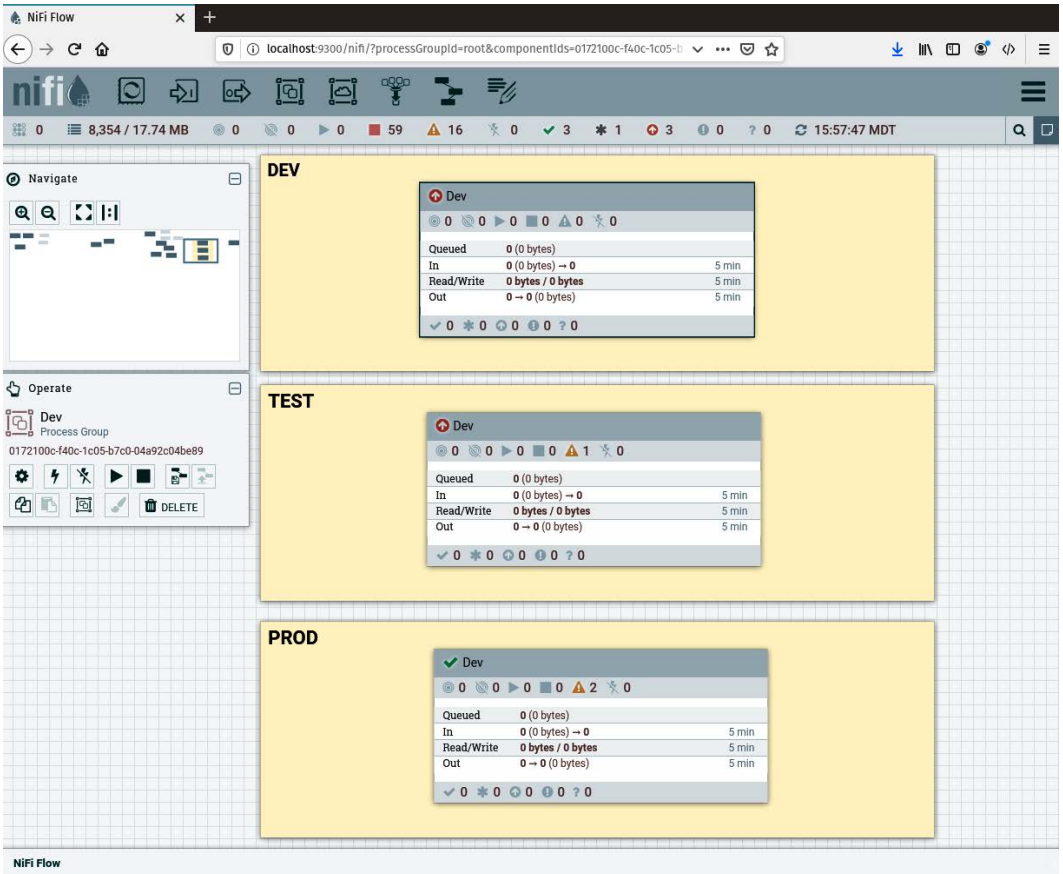

Figure 10.11 – A single NiFi instance working as DEV, TEST, and PROD

Notice in the preceding screenshot that only `PROD` has a green checkmark. The `DEV` environment created the processor group, then changes were committed, and they were brought into `TEST`. If any changes were made, they were committed, and the newest version was brought in to `PROD`. To improve the data pipeline later, you would bring the newest version into `DEV` and start the process over until `PROD` has the newest version as well.

While this will work, if you have the resources to build out a separate NiFi instance, you should.

Using the middle strategy

The middle strategy utilizes the NiFi registry but also adds a production NiFi instance. I have installed NiFi on another machine, separate from the one I have used through this book, that is also running the NiFi registry—this could also live on a separate machine.

After launching my new NiFi instance, I added the NiFi registry as shown in the following screenshot:

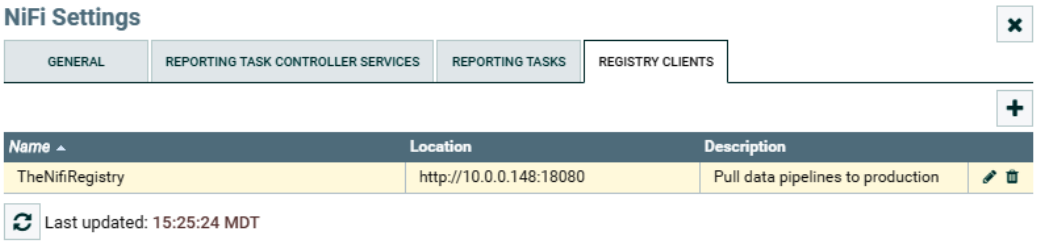

Figure 10.12 – Adding the NiFi registry to another NiFi instance

On the development machine, the registry was created using localhost. However, other machines can connect by specifying the IP address of the host machine. After reading it, the NiFi instance has access to all the versioned data pipelines.

Drag a processor group to the canvas and select **Import**. You can now select the processor group that has been promoted to production, as shown in the following screenshot:

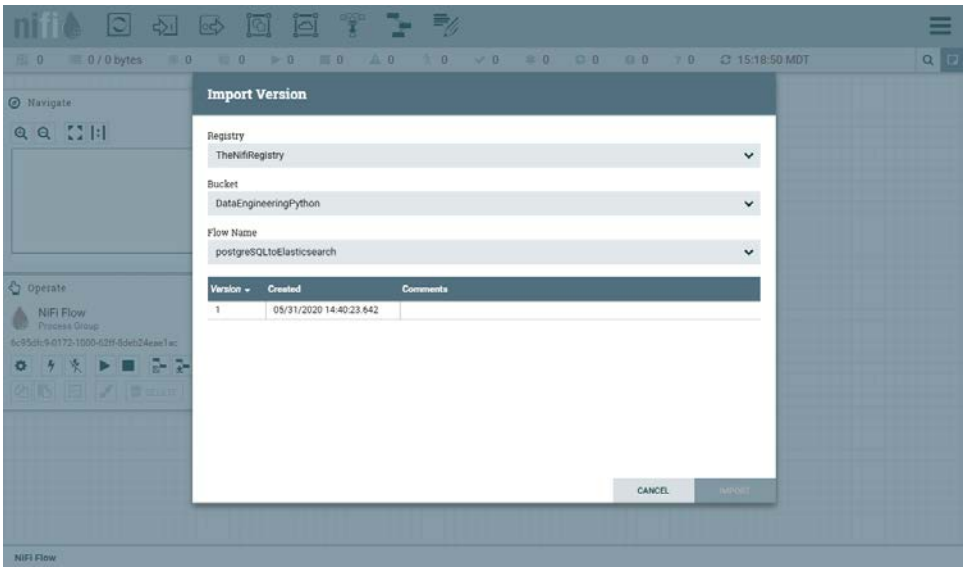

Figure 10.13 – Importing the processor group

Once you import the processor, it will come over with the variables that were defined in the development environment. You can overwrite the values of the variables. Once you change the variables, you will not need to do it again. You can make the changes in the development environment and update the production environment and the new variables will stay. The updated variables are shown in the following screenshot:

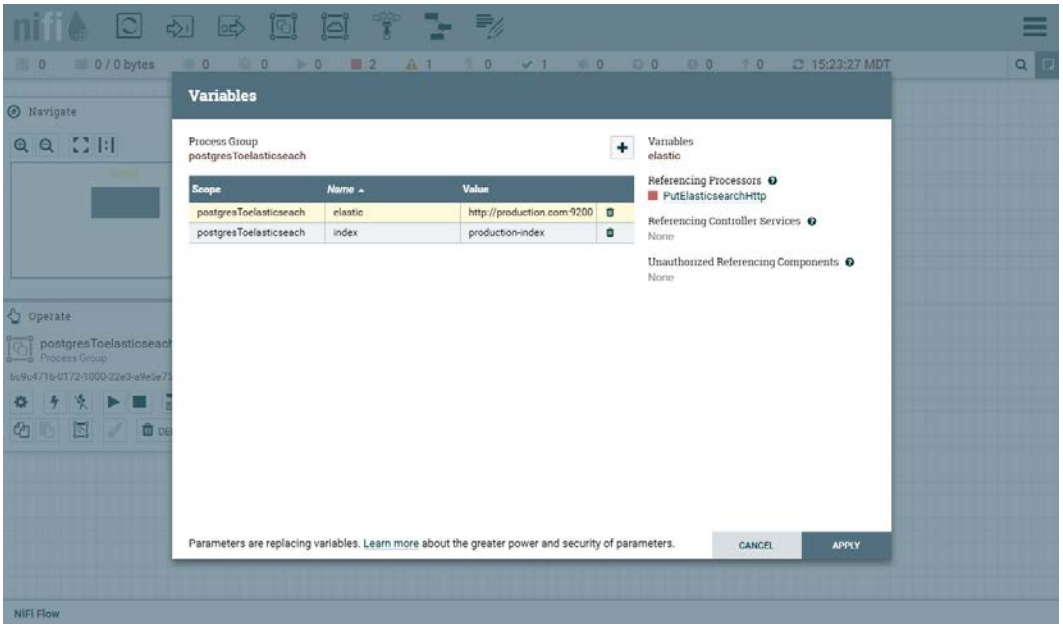

Figure 10.14 – Updating local variables for production

In the development environment, you can change the processor and commit the local changes. The production environment will now show that there is a new version available, as shown in the following screenshot:

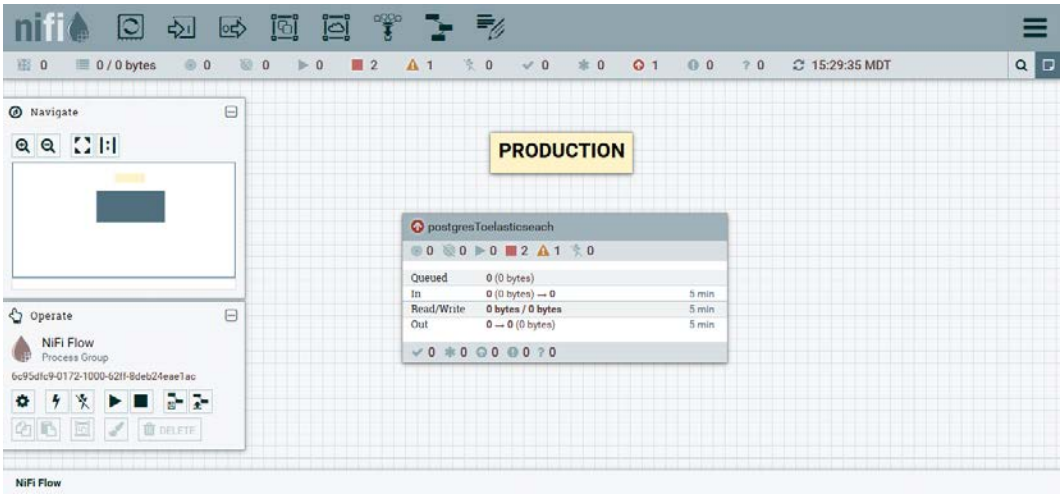

Figure 10.15 – Production is now no longer using the current version

You can right-click the processor group and select the new version. The following screenshot shows version 2:

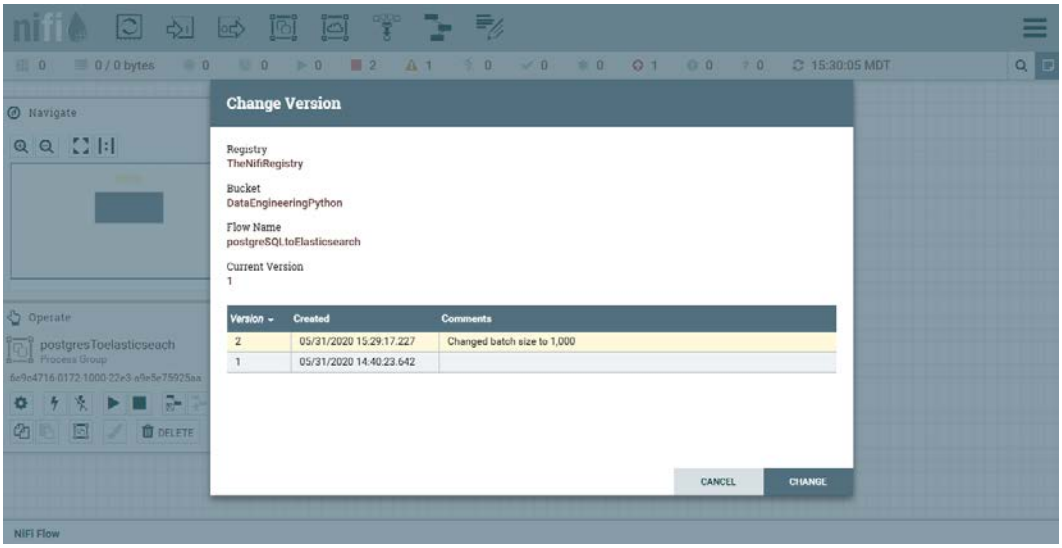

Figure 10.16 – A new version

After selecting the new version, the production environment is now up to date. The following screenshot shows the production environment. You can right-click on the processor group to see that the variable still points to the production values:

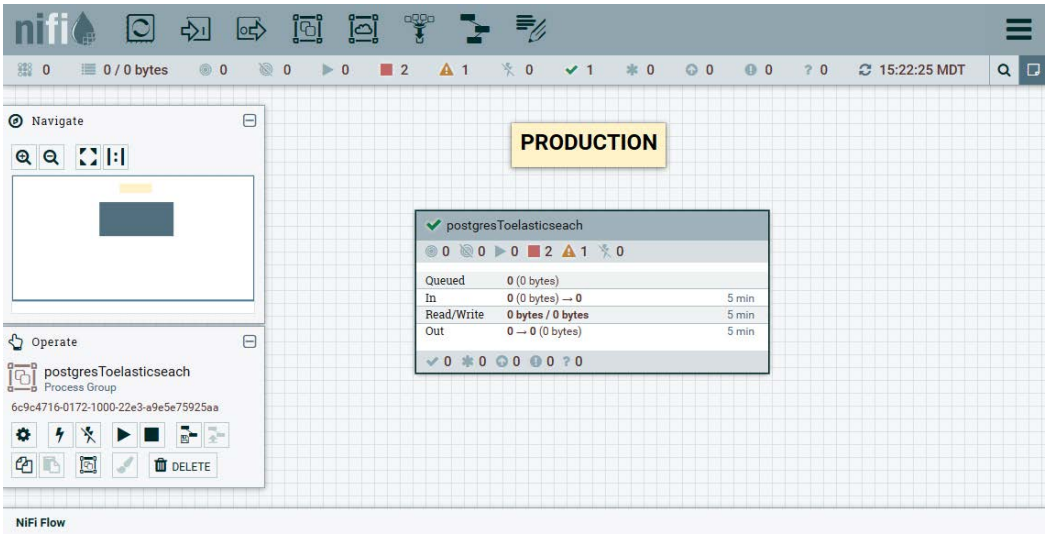

Figure 10.17 – Production is up to date

This strategy should work for most users' needs. In this example, I used development and production environments, but you can add `TEST` and use the same strategy here, just change the local variables to point to your test databases.

The preceding strategies used a single NiFi registry, but you can use a registry per environment.

Using multiple registries

A more advanced strategy for managing development, test, and production would be to use multiple NiFi registries. In this strategy, you would set up two NiFi registries—one for development and one for test and production. You would connect the development environment to the development registry and the test and production environments to the second registry.

When you have promoted a data pipeline to test, an administrator would use the NiFi CLI tools to export the data pipeline and import it in to the second NiFi registry. From there, you could test and promote it to development. You would import the version from the second registry to the production environment, just like you did in the middle strategy. This strategy makes mistakes much more difficult to handle as you cannot commit data pipelines to test and production without manually doing so. This is an excellent strategy but requires many more resources.

Summary

In this chapter, you learned how to finalize your data pipelines for deployment into production. By using processor groups for specific tasks, much like functions in code, you could reduce the duplication of processors. Using input and output ports, you connected multiple processor groups together. To deploy data pipelines, you learned how NiFi variables could be used to declare global and locally scoped variables.

In the next chapter, you will use all the skills you have learned in this section to create and deploy a production data pipeline.

11
Building a Production Data Pipeline

In this chapter, you will build a production data pipeline using the features and techniques that you have learned in this section of the book. The data pipeline will be broken into processor groups that perform a single task. Those groups will be version controlled and they will use the NiFi variable registry so that they can be deployed in a production environment.

In this chapter, we're going to cover the following main topics:

- Creating a test and production environment
- Building a production data pipeline
- Deploying a data pipeline in production

Creating a test and production environment

In this chapter, we will return to using PostgreSQL for both the extraction and loading of data. The data pipeline will require a test and production environment, each of which will have a staging and a warehouse table. To create the databases and tables, you will use **PgAdmin4**.

Creating the databases

To use PgAdmin4, perform the following steps:

1. Browse to `http://localhostw/pgadmin4/l`, enter your username and password, and then click the **Login** button. Once logged in, expand the server icon in the left panel.
2. To create the databases, right-click on the databases icon and select **Create | Database**. Name the database `test`.
3. Next, you will need to add the tables. To create the staging table, right-click on **Tables | Create | Table**. On the **General** tab, name the table `staging`. Then, select the **Columns** tab. Using the plus sign, create the fields shown in the following screenshot:

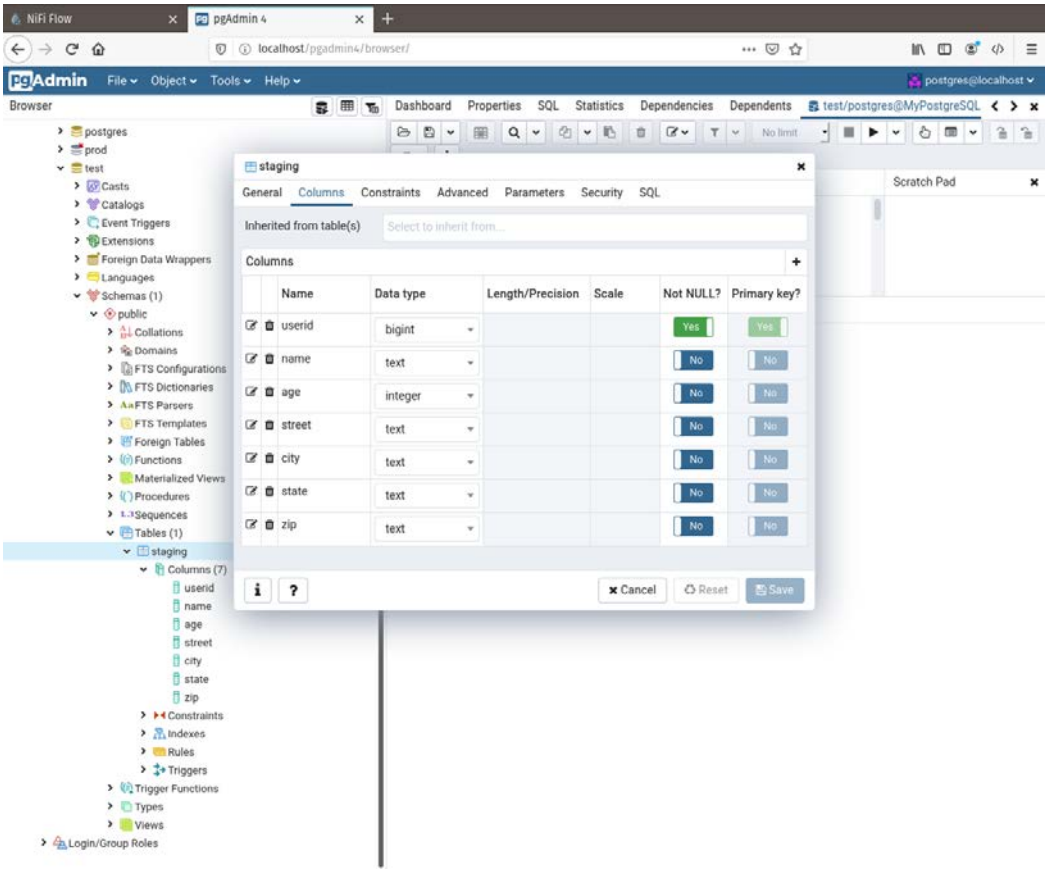

Figure 11.1 – The columns used in the staging table

4. Save the table when you are done. You will need to create this table once more for the test database and twice more for the production database. To save some time, you can use **CREATE Script** to do this for you. Right-click on the staging table, and then select **Scripts | CREATE Script**, as shown in the following screenshot:

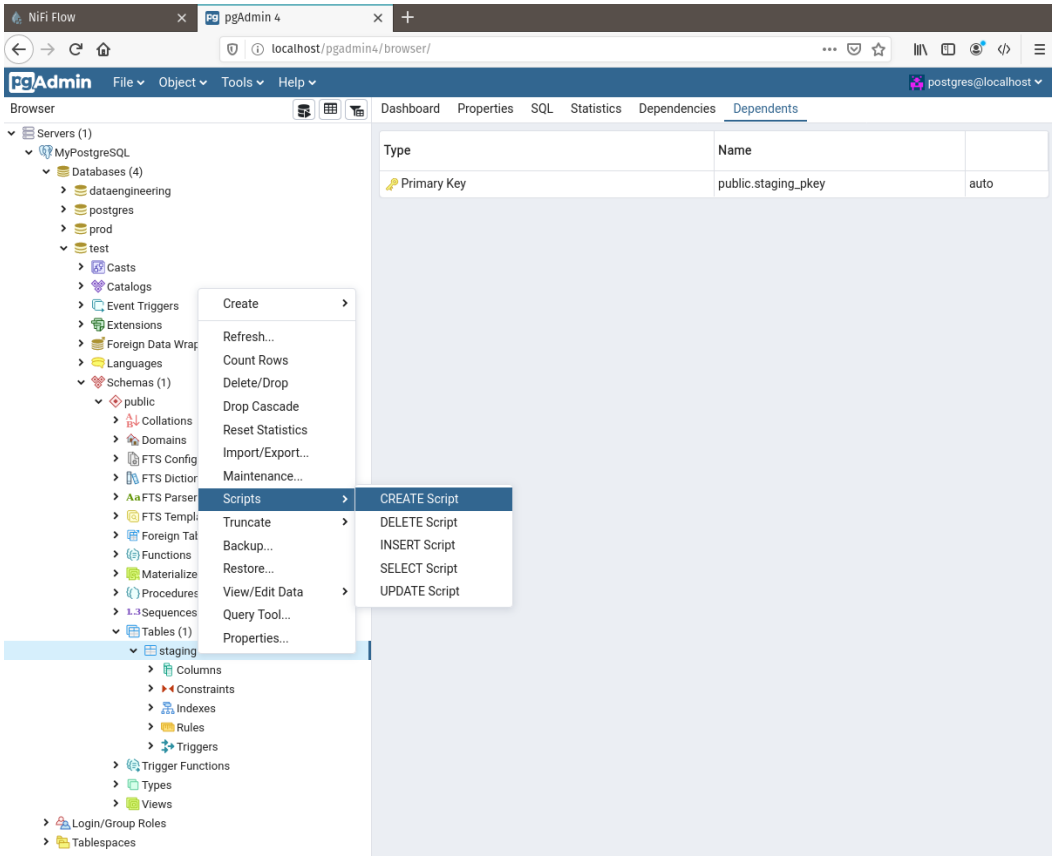

Figure 11.2 – Generating the CREATE script

5. A window will open in the main screen with the SQL required to generate the table. By changing the name from `staging` to `warehouse`, you can make the warehouse table in test, which will be identical to staging. Once you have made the change, click the play button in the toolbar.
6. Lastly, right-click on **Databases** and create a new database named `production`. Use the script to create both the tables.

Now that you have the tables created for the test and production environments, you will need a data lake.

Populating a data lake

A **data lake** is usually a place on disk where files are stored. Usually, you will find data lakes using Hadoop for the **Hadoop Distributed File System** (**HDFS**) and the other tools built on top of the Hadoop ecosystem. In this chapter, we will just drop files in a folder to simulate how reading from the data lake would work.

To create the data lake, you can use Python and the Faker library. Before you write the code, create a folder to act as the data lake. I have created a folder named `datalake` in my home directory.

To populate the data lake, you will need to write JSON files with information about an individual. This is similar to the JSON and CSV code you wrote in the first section of this book. The steps are as follows:

1. Import the libraries, set the data lake directory, and set `userid` to `1`. The `userid` variable is going to be a primary key, so we need it to be distinct – incrementing will do that for us:

```
from faker import Faker
import json
import os
os.chdir("/home/paulcrickard/datalake")
fake=Faker()
userid=1
```

2. Next, create a loop that generates a data object containing the user ID, name, age, street, city, state, and zip of a fake individual. The `fname` variable holds the first and last name of a person without a space in the middle. If you had a space, Linux would wrap the file in quotes:

```
for i in range(1000):
    name=fake.name()
    fname=name.replace(" ","-")+'.json'
    data={
        "userid":userid,
        "name":name,
        "age":fake.random_int(min=18, max=101, step=1),
        "street":fake.street_address(),
        "city":fake.city(),
        "state":fake.state(),
```

```
            "zip":fake.zipcode()
        }
```

3. Lastly, dump the JSON object and then write it to a file named after the person. Close the file and let the loop continue:

```
        datajson=json.dumps(data)
        output=open(fname,'w')
        userid+=1
        output.write(datajson)
        output.close()
```

Run the preceding code and you will have 1,000 JSON files in your data lake. Now you can start building the data pipeline.

Building a production data pipeline

The data pipeline you build will do the following:

- Read files from the data lake.
- Insert the files into staging.
- Validate the staging data.
- Move staging to the warehouse.

The final data pipeline will look like the following screenshot:

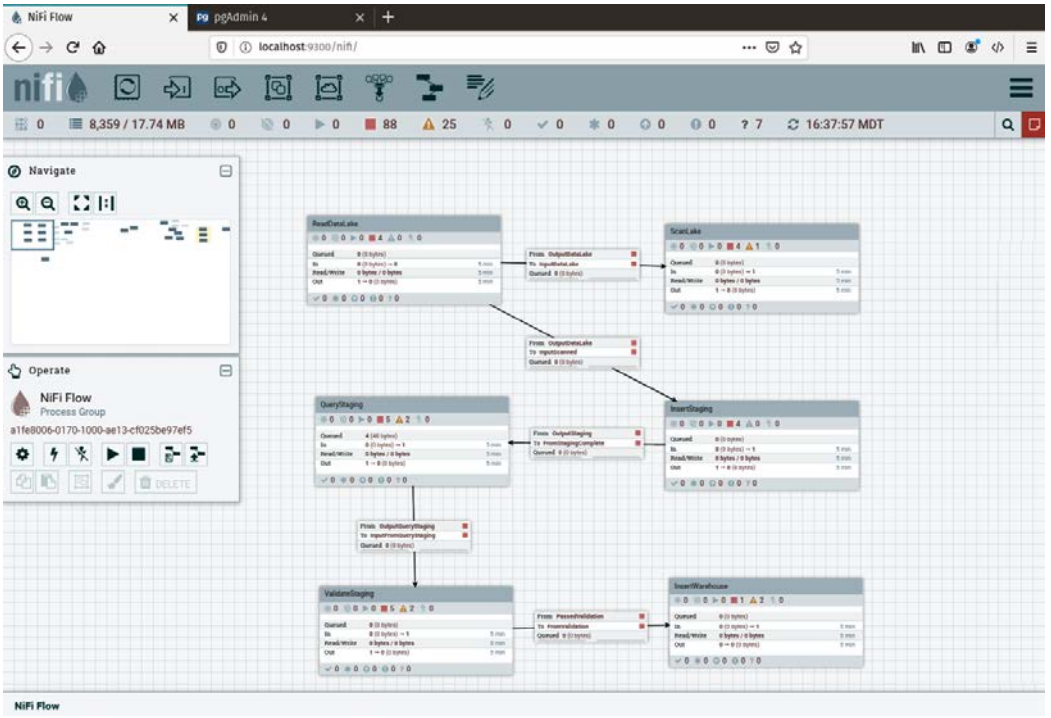

Figure 11.3 – The final version of the data pipeline

We will build the data pipeline processor group by processor group. The first processor group will read the data lake.

Reading the data lake

In the first section of this book, you read files from NiFi and will do the same here. This processor group will consist of three processors – `GetFile`, `EvaluateJsonPath`, and `UpdateCounter` – and an output port. Drag the processors and port to the canvas. In the following sections, you will configure them.

GetFile

The `GetFile` processor reads files from a folder, in this case, our data lake. If you were reading a data lake in Hadoop, you would switch out this processor for the `GetHDFS` processor. To configure the processor, specify the input directory; in my case, it is `/home/paulcrickard/datalake`. Make sure **Keep Source File** is set to **True**. If you wanted to move the processed files and drop them somewhere else, you could do this as well. Lastly, I have set **File Filter** to a regex pattern to match the JSON file extension – `^.*\.([jJ][sS][oO][nN]??)$`. If you leave the default, it will work, but if there are other files in the folder, NiFi will try to grab them and will fail.

EvaluateJsonPath

The `EvaluateJsonPath` processor will extract the fields from the JSON and put them into flowfile attributes. To do so, set the **Destination** property to **flowfile-attribute**. Leave the rest of the properties as the default. Using the plus sign, create a property for each field in the JSON. The configuration is shown in the following screenshot:

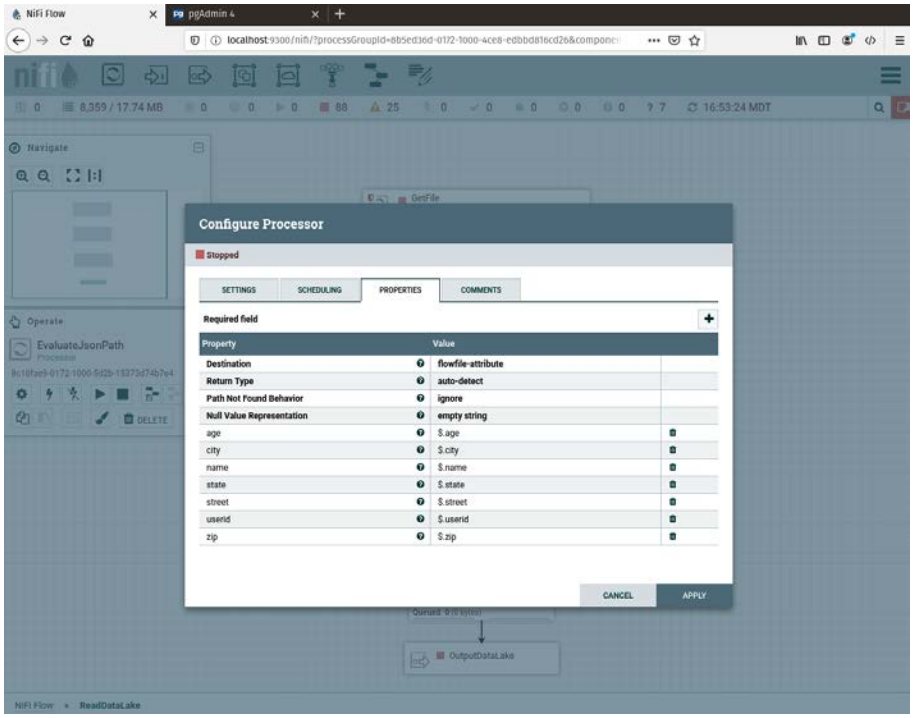

Figure 11.4 – The configuration for the EvaluateJsonPath processor

This would be enough to complete the task of reading from the data lake, but we will add one more processor for monitoring.

UpdateCounter

This processor allows you to create an increment counter. As flowfiles pass through, we can hold a count of how many are being processed. This processor does not manipulate or change any of our data, but will allow us to monitor the progress of the processor group. We will be able to see the number of FlowFiles that have moved through the processor. This is a more accurate way than using the GUI display, but it only shows the number of records in the last 5 minutes. To configure the processor, leave the **Delta** property set to `1` and set the **Counter Name** property to `datalakerecordsprocessed`.

To finish this section of the data pipeline, drag an output port to the canvas and name it `OutputDataLake`. Exit the processor group and right-click, select **Version**, and start version control. I set **Flow Name** to `ReadDataLake`, wrote a short description and version comments, and then performed a save.

> **NiFi-Registry**
>
> I have created a new bucket named `DataLake`. To create buckets, you can browse to the registry at `http://localhost:18080/nifi-registry/`. Click the wrench in the right corner and then click the **NEW BUCKET** button. Name and save the bucket.

The first processor group is complete. You can use this processor group any time you need to read from the data lake. The processor group will hand you every file with the fields extracted. If the data lake changed, you would only need to fix this one processor group to update all of your data pipelines.

Before continuing down the data pipeline, the next section will take a small diversion to show how you can attach other processor groups.

Scanning the data lake

The goal of the data pipeline is to read the data lake and put the data in the data warehouse. But let's assume there is another department at our company that needs to monitor the data lake for certain people – maybe VIP customers. Instead of building a new data pipeline, you can just add their task to the `ReadDataLake` processor group.

The `ScanLake` processor group has an input port that is connected to the output of the `ReadDataLake` processor. It uses the `ScanContent` processor attached to the `EvaluateJsonPath` processor, which is terminated at the `PutSlack` processor, as well as sending the data through to an output port. The flow is shown in the following screenshot:

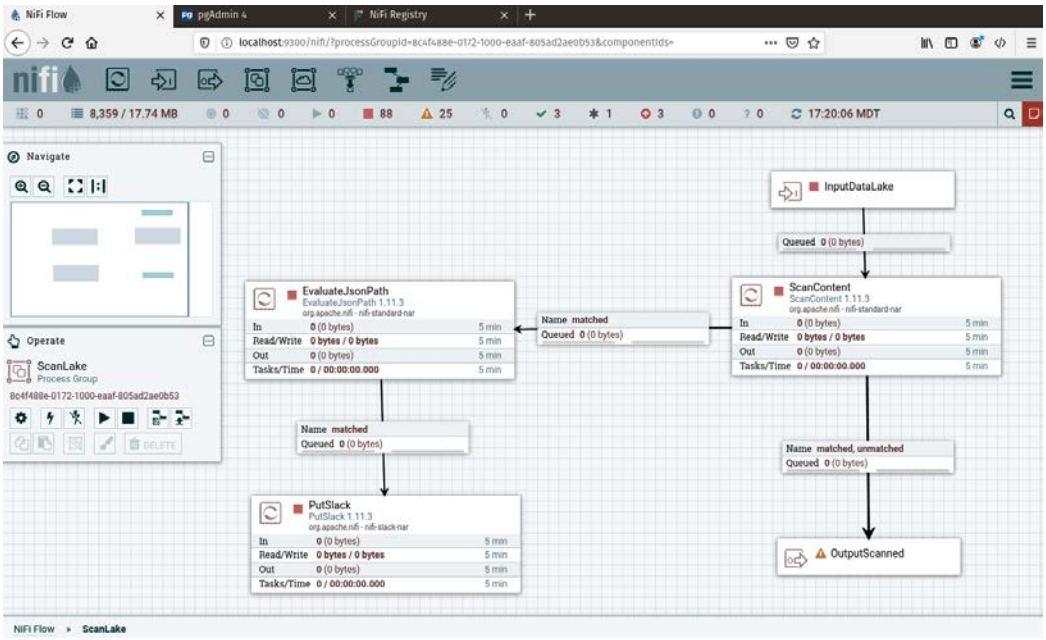

Figure 11.5 – The ScanLake processor group

The previous chapter used the `PutSlack` processor and you are already familiar with the `EvaluateJsonPath` processor. `ScanContent`, however, is a new processor. The `ScanContent` processor allows you to look at fields in the flowfile content and compare them to a dictionary file – a file with content on each line that you are looking for. I have put a single name in a file at `/home/paulcrickard/data.txt`. I configured the processor by setting the path as the value of the **Dictionary File** property. Now, when a file comes through that contains that name, I will get a message on Slack.

Inserting the data into staging

The data we read was from the data lake and will not be removed, so we do not need to take any intermediary steps, such as writing data to a file, as we would have done had the data been from a transactional database. But what we will do is place the data in a staging table to make sure that everything works as we expect before putting it in the data warehouse. To insert the data into staging only requires one processor, `PutSQL`.

PutSQL

The `PutSQL` processor will allow you to execute an `INSERT` or `UPDATE` operation on a database table. The processor allows you to specify the query in the contents of a flowfile, or you can hardcode the query to use as a property in the processor. For this example, I have hardcoded the query in the **SQL Statement** property, which is shown as follows:

```
INSERT INTO ${table} VALUES ('${userid}',
'${name}',${age},'${street}','${city}','${state}','${zip}');
```

The preceding query takes the attributes from the flowfile and passes them into the query, so while it is hardcoded, it will change based on the flowfiles it receives. You may have noticed that you have not used `${table}` in any of the `EvaluateJsonPath` processors. I have declared a variable using the NiFi registry and added it to the processor group scope. The value of the table will be `staging` for this test environment, but will change later when we deploy the data pipeline to production.

You will also need to add a **Java Database Connection** (**JDBC**) pool, which you have done in earlier chapters of this book. You can specify the batch size, the number of records to retrieve, and whether you want the processor to roll back on failure. Setting **Rollback on Failure** to **True** is how you can create atomicity in your transactions. If a single flowfile in a batch fails, the processor will stop and nothing else can continue.

I have connected the processor to another `UpdateCounter` processor. This processor creates and updates `InsertedStaging`. The counter should match `datalakerecordsprocessor` when everything has finished. The `UpdateCounter` processor connects to an output port named `OutputStaging`.

Querying the staging database

The next processor group is for querying the staging database. Now that the data has been loaded, we can query the database to make sure all the records have actually made it in. You could perform other validation steps or queries to see whether the results match what you would expect – if you have data analysts, they would be a good source of information for defining these queries. In the following sections, you will query the staging database and route the results based on whether it meets your criteria.

ExecuteSQLRecord

In the previous processor group, you used the `PutSQL` processor to insert data into the database, but in this processor group, you want to perform a `select` query. The `select` query is shown as follows:

```
select count(*) from ${table}
```

The preceding query is set as the value of the optional SQL `select` query property. The `${table}` is a NiFi variable registry variable assigned to the processor group and has a value of `staging`. You will need to define a JDBC connection and a record writer in the processor properties. The record writer is a JSON record set writer. The return value of the processor will be a JSON object with one field – `count`. This processor is sent to an `EvaluateJsonPath` processor to extract the count as `recordcount`. That processor is then sent to the next processor.

RouteOnAttribute

The `RouteOnAttribute` processor allows you to use expressions or values to define where a flowfile goes. To configure the processor, I have set the **Routing strategy** to **Route to Property name**. I have also created a property named `allrecords` and set the value to a NiFi expression, shown as follows:

```
${recordcount:ge( 1000 )}
```

The preceding expression evaluates the `recordcount` attribute to see whether it is greater than or equal to 1,000. If it is, it will route on this relationship. I have attached the output to an output port named `OutputQueryStaging`.

Validating the staging data

The previous processor group did some validation and you could stop there. However, Great Expectations is an excellent library for handling validation for you. You learned about Great Expectations in *Chapter 7, Features of a Production Pipeline*, but I will cover it quickly again here.

To use Great Expectations, you need to create a project folder. I have done that in the following code snippet and initialized Great Expectations:

```
mkdir staging
great_expectations init
```

You will be prompted to create your validation suite. Choose **Relational database (SQL)**, then **Postgres**, and provide the required information. The prompts will look like the following screenshot:

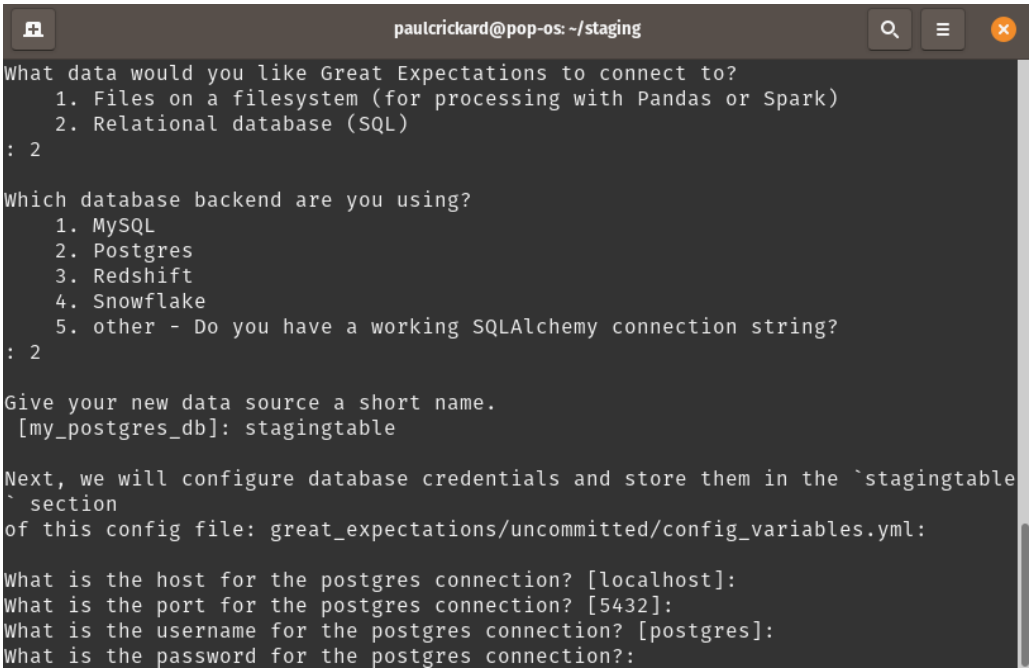

Figure 11.6 – Configuring Great Expectations to work with PostgreSQL

When it is finished, Great Expectations will attempt to connect to the database. If successful, it will provide the URL for your documents. Since the table is empty, it will not create a very detailed validation suite. You can edit the suite using the following command:

```
great_expectations suite edit staging.validation
```

This will launch a Jupyter notebook with the code for the suite. I have deleted one line that sets the number of rows between 0 and 0, as shown in the following screenshot:

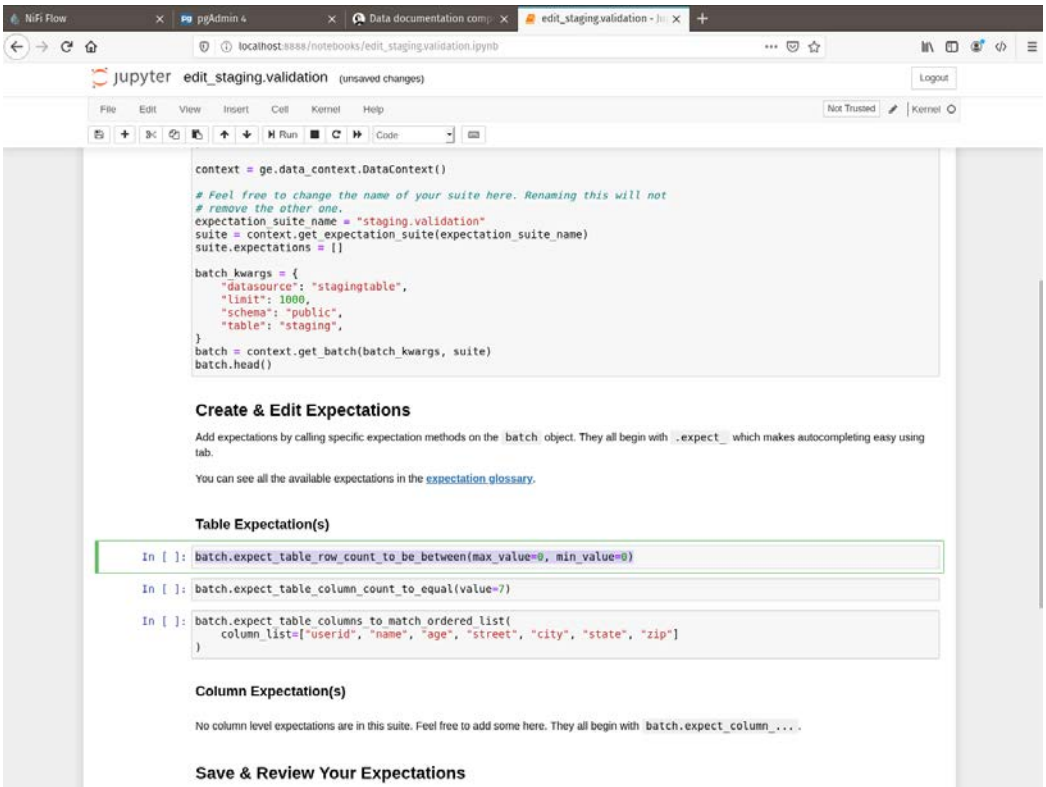

Figure 11.7 – Editing the Great Expectations suite

After deleting the highlighted line, run all the cells in the notebook. Now you can refresh your documents and you will see that the expectation on the row number is no longer part of the suite, as shown in the following screenshot:

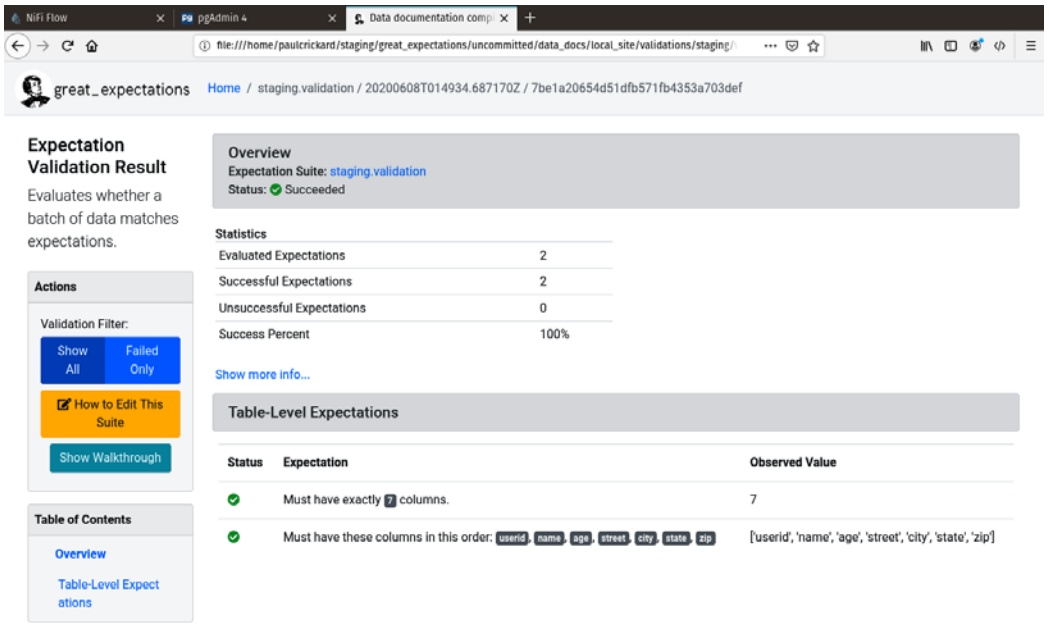

Figure 11.8 – Great Expectations documents for the suite

Now that the suite is complete, you need to generate a file that you can run to launch the validation. Use the following command to create a tap using the `staging.validation` suite and output the `sv.py` file:

```
great_expectations tap new staging.validation sv.py
```

Now you can run this file to validate the test database staging table.

The first processor receives flowfiles from an input port that is connected to the output port of the `QueryStaging` processor group. It connects to an `ExecuteStreamCommand` processor.

ExecuteStreamCommand

The `ExecuteStreamCommand` will execute a command and listen for output, streaming the results. Since the `sv.py` file only prints a single line and exits, there is no stream, but if your command had multiple outputs, the processor would grab them all as they were output.

To configure the processor, set the **Command Arguments** property to `sv.py`, the **Command Path** to **Python3**, and the working directory to the location of the `sv.py` file.

The processor connects to an `EvaluateJsonPath` processor that extracts `$.result` and sends it to a `RouteOnAttribute` processor. I have configured a single property and accorded it the value `pass`:

```
${result:startsWith('pass')}
```

The preceding expression checks the result attribute to see whether it matches `pass`. If so, the processor sends the flowfile to an output port.

Insert Warehouse

You have made it to the last processor group – **Insert Warehouse**. The data has been staged and validated successfully and is ready to move to the warehouse. This processor group uses an `ExecuteSQLRecord` and a `PutSQL` processor.

ExecuteSQLRecord

`ExecuteSQLProcessor` performs a select operation on the staging table. It has a variable table defined in the NiFi variable registry pointing to staging. The query is a `select *` query, as shown:

```
select * from ${table}
```

This query is the value of the SQL `select` query property. You will need to set up a `Database Pooling Connection` service and a `Record Writer` service. `Record Writer` will be a `JsonRecordSetWriter` and you will need to make sure that you set **Output Grouping** to **One Line per Object**. This processor sends the output to the `SplitText` processor, which connects to the `EvalueJsonPath` processor, which is a direct copy of the one from the `ReadDataLake` processor group that connects to the final `PutSQL` processor.

PutSQL

The `PutSQL` processor puts all of the data from the `staging` table into the final data `warehouse` table. You can configure the batch size and the rollback on failure properties. I have set the SQL Statement property to the same as when it was inserted into `staging`, except the variable for the table has been changed to `warehouse` and we set it to `warehouse` in the NiFi variable registry. The query is as follows:

```
INSERT INTO ${warehouse} VALUES ('${userid}',
'${name}',${age},'${street}','${city}','${state}','${zip}');
```

I have terminated the processor for all relationships as this is the end of the data pipeline. If you start all the processor groups, you will have data in your `staging` and `warehouse` tables. You can check your counters to see whether the records processed are the same as the number of records inserted. If everything worked correctly, you can now deploy your data pipeline to production.

Deploying a data pipeline in production

In the previous chapter, you learned how to deploy data to production, so I will not go into any great depth here, but merely provide a review. To put the new data pipeline into production, perform the following steps:

1. Browse to your production NiFi instance. I have another instance of NiFi running on port `8080` on localhost.
2. Drag and drop processor groups to the canvas and select **Import**. Choose the latest version of the processor groups you just built.
3. Modify the variables on the processor groups to point to the database production. The table names can stay the same.

You can then run the data pipeline and you will see that the data is populated in the production database `staging` and `warehouse` tables.

The data pipeline you just built read files from a data lake, put them into a database table, ran a query to validate the table, and then inserted them into the warehouse. You could have built this data pipeline with a handful of processors and been done, but when you build for production, you will need to provide error checking and monitoring. Spending the time up front to build your data pipelines properly will save you a lot of time when something changes or breaks in production. You will be well positioned to debug and modify your data pipeline.

Summary

In this chapter, you learned how to build and deploy a production data pipeline. You learned how to create `TEST` and `PRODUCTION` environments and built the data pipeline in `TEST`. You used the filesystem as a sample data lake and learned how you would read files from the lake and monitor them as they were processed. Instead of loading data into the data warehouse, this chapter taught you how to use a staging database to hold the data so that it could be validated before being loaded into the data warehouse. Using Great Expectations, you were able to build a validation processor group that would scan the staging database to determine whether the data was ready to be loaded into the data warehouse. Lastly, you learned how to deploy the data pipeline into `PRODUCTION`. With these skills, you can now fully build, test, and deploy production batch data pipelines.

In the next chapter, you will learn how to build Apache Kafka clusters. Using Kafka, you will begin to learn how to process data streams. This data is usually near real time, as opposed to the batch processing you have been currently working with. You will install and configure the cluster to run on a single machine, or multiple devices if you have them.

Section 3: Beyond Batch – Building Real-Time Data Pipelines

In this section, you will learn about the differences between batch processing – what you have currently been doing – and stream processing. You will learn about a new set of tools that allow you to stream and process data in real time. First, you will learn how to build an Apache Kafka cluster to stream real-time data. To process this data, you will use an Apache Spark cluster that you will build and deploy. Lastly, you will learn two more advanced NiFi topics – how to stream data to NiFi from an Internet of Things device using MiNiFi, and how to cluster NiFi for more processing power.

This section comprises the following chapters:

- *Chapter 12*, *Building an Apache Kafka Cluster*
- *Chapter 13*, *Streaming Data with Kafka*
- *Chapter 14*, *Data Processing with Apache Spark*
- *Chapter 15*, *Real-Time Edge Data – Kafka, Spark, and MiNiFi*

12
Building a Kafka Cluster

In this chapter, you will move beyond batch processing – running queries on a complete set of data – and learn about the tools used in stream processing. In stream processing, the data may be infinite and incomplete at the time of a query. One of the leading tools in handling streaming data is Apache Kafka. Kafka is a tool that allows you to send data in real time to topics. These topics can be read by consumers who process the data. This chapter will teach you how to build a three-node Apache Kafka cluster. You will also learn how to create and send messages (**produce**) and read data from topics (**consume**).

In this chapter, we're going to cover the following main topics:

- Creating ZooKeeper and Kafka clusters
- Testing the Kafka cluster

Creating ZooKeeper and Kafka clusters

Most tutorials on running applications that can be distributed often only show how to run a single node and then you are left wondering how you would run this in production. In this section, you will build a three-node ZooKeeper and Kafka cluster. It will run on a single machine. However, I will split each instance into its own folder and each folder simulates a server. The only modification when running on different servers would be to change localhost to the server IP.

The next chapter will go into detail on the topic of Apache Kafka, but for now it is enough to understand that Kafka is a tool for building real-time data streams. Kafka was developed at LinkedIn and is now an Apache project. You can find Kafka on the web at `http://kafka.apache.org`. The website is shown in the following screenshot:

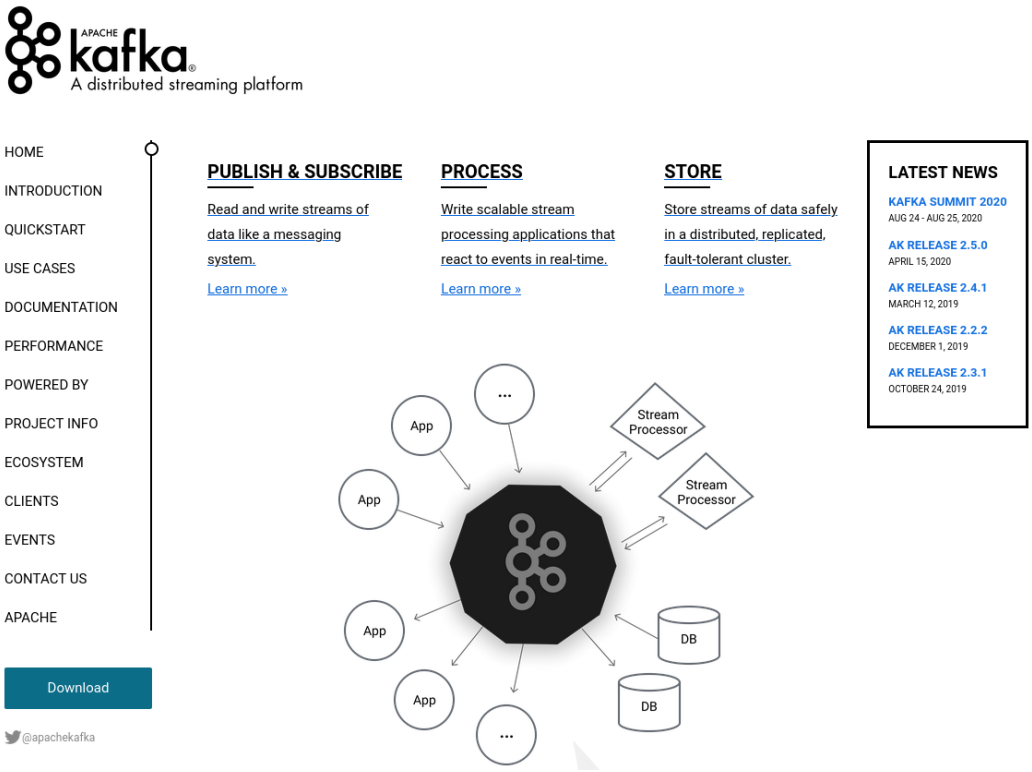

Figure 12.1 – Apache Kafka website

Kafka requires another application, ZooKeeper, to manage information about the cluster, to handle discovery, and to elect leaders. You can install and build a ZooKeeper cluster on your own, but for this example, you will use the ZooKeeper scripts provided by Kafka. To learn more about ZooKeeper, you can find it at `http://zookeeper.apache.org`. The website is shown in the following screenshot:

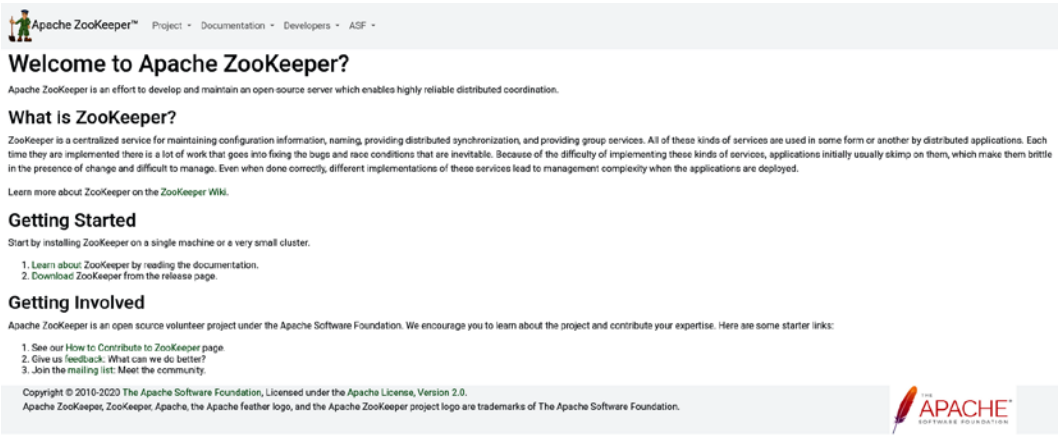

Figure 12.2 – The Apache ZooKeeper website

The following section will walk you through building the cluster.

Downloading Kafka and setting up the environment

You can download Apache Kafka from the website under the **Downloads** section – which is useful if you want a previous version – or you can use `wget` to download it from the command line. From your home directory, run the following commands:

```
Wget https://downloads.apache.org/kafka/2.5.0/kafka_2.12-2.5.0.tgz
tar -xvzf kafka_2.12-2.5.0.tgz
```

The preceding commands download the current Kafka version and extract it into the current directory. Because you will run three nodes, you will need to create three separate folders for Kafka. Use the following commands to create the directories:

```
cp kafka_2.12-2.5.0 kafka_1
cp kafka_2.12-2.5.0 kafka_2
cp kafka_2.12-2.5.0 kafka_3
```