ROUTING AND TRANSFORMATION

Previous chapters covered the construction of databases, Data Lakes, and Data Warehouses, the foundational elements of a data platform, all from within Kubernetes, demonstrating a robust distributed services platform. This chapter focuses on the collection, extraction, movement, and processing of data, rounding out the majority of functionality required for any data-centric application. The ability to efficiently extract, transform, and load data from one diverse system into another is essential in harnessing the explosive growth of data from consumer and industrial IoT, social media, and digital transformation occurring in many organizations. The ability to quickly construct routes that move, transform, and process data is vital in leveraging the ever-advancing, data-driven trends such as Machine Learning based AI, technologies particularly hungry for large quantities of processed data. An effective data platform provides all the generalized mechanisms needed to extract, transform, and load data across data management systems and offers an application layer for supporting specialized processing and custom business logic.
This chapter extends the Kubernetes-based data platform built up from previous chapters with ETL/ELT (extract, transform, load) and FaaS (functions as a service, also known as Serverless) functionality. Techniques and technologies focused on ETL have been maturing for many years. Data engineers can now combine ETL with Serverless platforms, quickly developing, integrating, and deploying directly into the data pipeline.

ETL and Data Processing

This chapter introduces two new technologies into Kubernetes: Apache NiFi and OpenFaaS, demonstrating a purely open source method of extracting, loading, transforming, and processing data without the need for domain-specific languages and complicated configuration files. Apache NiFi brings hundreds of prebuilt data “processors” for the extraction and loading of data to and from nearly any standard network protocol or API implementation, along with the ability to transform it to almost any desired form. OpenFaaS is a vendor-neutral approach to the concept of Serverless/FaaS (functions as a services); in this case, FaaS allows for the limitless extensibility of the ETL pipeline through custom, highly focused code, developed in any language.

Figure 9-1 depicts an ETL, data processing, and visualization demonstration developed throughout this chapter. The end goal is to visualize the range of sentiment expressed over some time on a specific set of Twitter messages. The following sections install OpenFaaS and deploy a prepacked sentiment analysis Function. Later sections introduce Apache NiFi and configure directed graphs of data routing and transformation from Twitter to Kafka and out of Kafka to the OpenFaaS sentiment analysis Function, finally recording the results in Elasticsearch, exposed for analysis by a Jupyter Notebook. While none of these technologies requires Kubernetes, this demonstration aims to illuminate the advantages of a unified control plane, networking, monitoring, and the extensibility of its container-based application platform capabilities.
Some technologies in this book, namely, databases, may sacrifice a degree of performance when running over abstracted infrastructure; however, some applications will find this an acceptable trade-off for the reduction of technical debt incurred when managing many systems with thoroughly distinctive dependencies. Depending on the size of the organization or budgetary concerns of the project, it may not be feasible to employ infrastructure expertise for each enterprise-focused technology such as Apache NiFi, Kafka, and Elasticsearch. This book hopes to demonstrate these technologies on a scale nearly any project may utilize by leveraging Kubernetes, from a startup proof of concept to a web-scale social network.

**Development Environment**

The following exercises continue to utilize the inexpensive Hetzner cluster mentioned in Chapter 6, including one CX21 (2 vCPU/8G RAM/40G SSD) for the Kubernetes master node and four CX41 (4 vCPU/16G RAM/160G...
SSD) instances for worker nodes, yet any equivalent infrastructure will accommodate. Additionally, this chapter leverages applications and cluster configurations installed in Chapters 3, 5, and 6; see Table 7-1 from Chapter 7. This chapter requires Ingress, Cert Manager, Storage, and Monitoring configured in Chapter 3; Namespace, Zookeeper, and Kafka from Chapter 5; and Elasticsearch, Kibana, Keycloak, and JupyterHub from Chapter 6.

This chapter organizes configuration manifests for the cluster named `dev5`, under the project folder `cluster-apk8s-dev5`.

**Serverless**

The concept of Serverless, otherwise known as FaaS (functions as a service), has continued to mature and fill the need for the streamlined deployment of small units of functional code. The difference between a monolithic application, a microservice, and a (Serverless) Function resides in the implementation, operational infrastructure, and context of the broader architecture. For example, many microservice and function-based architectures still rely on monolithic databases. The term Serverless implies that the developer should require little or no concern over the server-side implementation. Serverless, or FaaS (functions as a service), aims to abstract away nearly all aspects of integration, deployment, and runtime operations, leaving functional business logic as the only responsibility of the developer.

Cloud vendors market the central appeal of Serverless technology, the ability to allow developers to focus solely on business logic, abstracting and managing infrastructure, operating systems, and runtime and application layers. The major cloud vendors’ offerings include Amazon’s
AWS Lambda, Microsoft’s Azure Functions, Google Cloud Functions, and IBM Cloud Functions. These products can significantly reduce time to market and technical debt for many organizations, albeit at the cost of vendor lock-in. Organizations already invested in Kubernetes, however, can take advantage of the growing number of open source, vendor-neutral Serverless platforms such as Apache OpenWhisk, Kubeless, and OpenFaaS. Knative is a popular choice for those looking to develop a custom Serverless platform.

This chapter demonstrates OpenFaaS for use in an example ETL application and how Serverless/functions as a service can make an excellent addition to any data platform.

**OpenFaaS**

OpenFaaS is a stable, well-maintained, Serverless application platform used by a growing number of organizations. OpenFaaS runs nearly anywhere, yet integrates well with Kubernetes, supporting languages such as Go, Java, Python, PHP, Rust, Perl, C#, and Ruby, along with application platforms including Express.js, Django, and ASP.NET Core. OpenFaaS supports custom-built containers for wrapping powerful binaries such as FFmpeg and ImageMagick.

---

1. [https://aws.amazon.com/lambda/](https://aws.amazon.com/lambda/)
2. [https://azure.microsoft.com/en-us/services/functions/](https://azure.microsoft.com/en-us/services/functions/)
3. [https://cloud.google.com/functions/docs/](https://cloud.google.com/functions/docs/)
4. [www.ibm.com/cloud/functions](www.ibm.com/cloud/functions)
5. [https://openwhisk.apache.org/](https://openwhisk.apache.org/)
6. [https://kubeless.io/](https://kubeless.io/)
7. [www.openfaas.com/](www.openfaas.com/)
8. [https://knative.dev/](https://knative.dev/)
9. [https://ffmpeg.org/](https://ffmpeg.org/)
10. [https://imagemagick.org/](https://imagemagick.org/)
The management of containerized workloads is Kubernetes’s central capability; however, a platform such as OpenFaaS provides a formal tooling and operations framework for the development, cataloging, integration, deployment, scaling, and monitoring of workloads expressed as Functions.

Serverless/functions as a service are a natural fit for ETL and data processing pipelines, as demonstrated later in this chapter.

**Install OpenFaaS**

Update Helm with the OpenFaaS repository:

```bash
$ helm repo add openfaas 
  https://openfaas.github.io/faas-netes/
$ helm repo update
```

Next, use Helm to install the OpenFaaS gateway into the data Namespace with the argument `--namespace data`. The OpenFaaS gateway can set Functions to run in an alternate Namespace, but for this example, use the data Namespace by setting `functionNamespace=data`. OpenFaaS is capable of utilizing Kubernetes Node Ports and load balancers; however, this example uses Ingress to expose the deployed Functions by setting `exposeServices=false` and `ingress.enabled=true`. Lastly, set the option `generateBasicAuth=true` to protect the Ingress exposed gateway user interface with Basic Authentication:

```bash
$ helm upgrade apk8s-data-openfaas -install 
  openfaas/openfaas 
  --namespace data 
  --set functionNamespace=data 
  --set exposeServices=false 
  --set ingress.enabled=true 
  --set generateBasicAuth=true
```

[11]https://kubernetes.io/docs/concepts/services-networking/service/
#publishing-services-service-types
Create the directory `cluster-apk8s-dev5/003-data/120-openfaas`. Within the new `120-openfaas` directory, create a file named `50-ingress.yml` from Listing 9-1.

**Listing 9-1. OpenFaaS Ingress**

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: faas
  namespace: data
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
spec:
  rules:
  - host: faas.data.dev5.apk8s.dev
    http:
      paths:
      - backend:
        serviceName: gateway
        servicePort: 8080
        path: /
    tls:
    - hosts:
      - faas.data.dev5.apk8s.dev
      secretName: faas-data-production-tls

Apply the OpenFaaS Ingress configuration:

```
After successfully applying Ingress, retrieve the Basic Authentication credentials generated by the OpenFaaS Helm installation with the following command:

```
$ echo $(kubectl -n data get secret basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode)
```

The previous command returns the Basic Auth password used to log in to the OpenFaaS UI portal; the username is admin. Browse to the ingress URL, in this example (see Figure 9-2):

https://faas.data.dev5.apk8s.dev.

![OpenFaaS UI portal](Image)

**Figure 9-2. OpenFaaS UI portal**

The OpenFaaS UI portal is a convenient web-based, visual interface for installing and managing Functions. However, the CLI utility faas-cli is often the preferred method of interacting with OpenFaaS, capable of supporting all aspects of developing, testing, deploying, and administering and automating functions as a service. Install the OpenFaaS CLI on a local workstation:

```
$ curl -sLSf https://cli.openfaas.com | sudo sh
```
Execute the OpenFaaS CLI for a top-level list of commands:

```bash
$ faas-cli
```

While this book recommends the CLI utility for regular use, the following section uses the web-based interface as a simple visual demonstration of installing and interacting with an OpenFaaS Function.

## Install Sentiment Analysis

Sentiment Analysis,\(^\text{12}\) otherwise known as emotion recognition\(^\text{13}\) or opinion mining,\(^\text{14}\) is a form of natural language processing (NLP). NLP applies linguistics, artificial intelligence, and information engineering to natural (human) languages. This section deploys a prebuilt OpenFaaS Function container,\(^\text{15}\) implementing the Python library TextBlob\(^\text{16}\) to perform Sentiment Analysis on one or more sentences of raw text. This chapter later uses the deployed Sentiment Analysis Function to analyze a real-time stream of Twitter messages tagged with keywords related to COVID-19.

Browse to the OpenFaaS UI portal set up in the previous section ([https://faas.data.dev5.apk8s.dev](https://faas.data.dev5.apk8s.dev)) and click the DEPLOY NEW FUNCTION button in the center of the screen. Next, use the Search for

\(^{12}\)Gupta, Shashank. “Sentiment Analysis: Concept, Analysis and Applications.” Medium, January 19, 2018. [https://towardsdatascience.com/sentiment-analysis-concept-analysis-and-applications-6c94d6f58c17](https://towardsdatascience.com/sentiment-analysis-concept-analysis-and-applications-6c94d6f58c17).

\(^{13}\)Kołakowska, Agata, Agnieszka Landowska, Mariusz Szwoch, Wioleta Szwoch, and Michał Wróbel. “Emotion Recognition and Its Applications.” Advances in Intelligent Systems and Computing 300 (July 1, 2014): 51–62. [https://doi.org/10.1007/978-3-319-08491-6_5](https://doi.org/10.1007/978-3-319-08491-6_5).

\(^{14}\)Analytics Vidhya. “A NLP Approach to Mining Online Reviews Using Topic Modeling,” October 16, 2018. [www.analyticsvidhya.com/blog/2018/10/mining-online-reviews-topic-modeling-lda/](http://www.analyticsvidhya.com/blog/2018/10/mining-online-reviews-topic-modeling-lda/).

\(^{15}\)https://github.com/openfaas/faas/tree/master/sample-functions/SentimentAnalysis

\(^{16}\)https://textblob.readthedocs.io/en/dev/
Function feature and search for the term SentimentAnalysis as shown in Figure 9-3, select the Function SentimentAnalysis, and click DEPLOY on the bottom left of the dialog.

![Image of Deploy A New Function]

**Figure 9-3.** OpenFaaS Deploy the prebuilt Sentiment Analysis Function

After deploying the OpenFaaS Sentiment Analysis Function, select it from the left-hand navigation. The UI portal displays the Status, Replicas, Invocation count, image, and URL for the Function in the upper section of the page (see Figure 9-4). The URL is the publicly exposed endpoint. The OpenFaaS gateway does not protect function endpoints by default,\(^\text{17}\) security is the responsibility of the Function itself. OpenFaaS documentation walks through developing custom Function authentication by implementing HMAC security using Kubernetes Secrets.\(^\text{18}\)

\(^{17}\)https://docs.openfaas.com/reference/authentication/#for-functions

\(^{18}\)https://github.com/openfaas/workshop/blob/master/lab11.md
The OpenFaaS UI portal provides a convenient web form for testing the deployed Function under the Invoke function section shown in Figure 9-4. Alternatively, invoke the Sentiment Analysis Function using `faas-cli` utility installed in the previous section:

```
$ echo "Kubernetes is easy" | faas-cli invoke \
    sentimentanalysis -g https://faas.data.dev5.apk8s.dev/
```

Finally, test public access to the new Function with cURL:

```
$ curl -X POST -d "People are kind" \
    https://faas.data.dev5.apk8s.dev/function/sentimentanalysis
```
Example output:
{"polarity": 0.6, "sentence_count": 1, "subjectivity": 0.9}

The OpenFaaS Sentiment Analysis Function is an excellent example of a focused, self-contained bit of processing logic deployed and managed by OpenFaaS atop Kubernetes. The OpenFaaS documentation contains a well-written set of tutorials on building, testing, and implementing Functions. Functions are a great way to extend the data platform developed in this book continuously. The next section covers Apache NiFi for ETL types of operations and incorporates the use of the Sentiment Analysis Function as part of an example data processing flow (see Figure 9-5).

**ETL**

The practice of ETL (extract, transform, load) dates back to the 1970s. The need to pull data from one source and transform it for use by another is a timeless problem, and today there is a wealth of existing techniques and technologies to address it. Pentaho, Talend, CloverETL, and JasperETL are a few commercial products with limited open source, community-driven options. However, ETL is such a common problem

---

19 https://github.com/openfaas/workshop
20 Health Catalyst. “Healthcare Information Systems: Past, Present, Future,” May 20, 2014. www.healthcatalyst.com/insights/healthcare-information-systems-past-present-future.
21 https://wiki.pentaho.com/
22 www.talend.com/products/talend-open-studio
23 www.cloverdx.com/
24 https://community.jaspersoft.com/project/jaspersoft-ett
that new approaches and generalized solutions such as the open source (vendor-neutral) Apache NiFi have been gaining popularity for ease of use and a simplistic, intuitive approach to data collection, routing, and transformation.

Apache NiFi

Apache NiFi is the data ingestion front-end of choice for the data-centric platform described in this book. “Apache NiFi supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.”

NiFi ships with nearly 300 unique data processors usable for the collection, transformation, processing, and modeling of data from sources as diverse as Twitter, SMTP, HDFS, Redis, UDP, HBase, and HTTP API endpoints.

At the time of this book’s publication, there is little official documentation and support for running NiFi within Kubernetes. However, the NiFi maintainers are aware of the rapid growth in demand for first-class Kubernetes support, and readers should expect significant contributions to this effort in the coming years.

The following sections install a multi-node Apache NiFi cluster in Kubernetes and demonstrate a flow of data extraction from Twitter with transformation, processing, and utilizing OpenFaaS, Kafka, and Elasticsearch, as shown in Figure 9-5.

25 https://nifi.apache.org/
26 https://nifi.apache.org/docs.html
Install Apache NiFi

This section installs Apache NiFi with three Kubernetes resources consisting of a Headless Service, StatefulSet, and Ingress. Review the “Development Environment” section earlier in this chapter for requirements, including Ingress Nginx, Ceph storage, Cert Manager, and Apache Zookeeper.

Although this book avoids Helm installations at times, in favor of a more verbose representation of concepts through hand-crafted manifests, readers should also consider the Apache NiFi Helm chart by Cetic.27

27 https://github.com/cetic/helm-nifi
Create the directory `cluster-apk8s-dev5/003-data/060-nifi`. Within the new `060-nifi` directory, create a file named `10-service-headless.yml` from Listing 9-2. The StatefulSet defined in the following calls for two replicas of the Pod `nifi` running an `apache/nifi:1.9.2` container. Each NiFi instance in the cluster requires some custom configuration on boot. Rather than allow the standard startup script to run, the `command:` section invokes Bash and pipes in a script to customize a few properties based on the hostname Kubernetes assigns to the Pod.

**Listing 9-2.** NiFi Headless Service

```
apiVersion: v1
kind: Service
metadata:
  name: nifi
  namespace: data
labels:
  app: nifi
annotations:
  service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
spec:
  type: ClusterIP
  clusterIP: None
  selector:
    app: nifi
  ports:
  - port: 8080
    name: http
  - port: 6007
    name: cluster
```
Apply the NiFi Headless Service configuration:

```
$ kubectl apply -f 10-service-headless.yml
```

Next, create a StatefulSet configuration for NiFi in a file named 40-statefulset.yml from Listing 9-3.

### Listing 9-3. NiFi StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nifi
  namespace: data
  labels:
    app: nifi
spec:
  replicas: 2
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      app: nifi
    serviceName: nifi
template:
  metadata:
    labels:
      app: nifi
  spec:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
```

352
matchExpressions:
  - key: "app"
    operator: In
    values:
      - "nifi"
  topologyKey: "kubernetes.io/hostname"

containers:
  - name: nifi
    imagePullPolicy: IfNotPresent
    image: apache/nifi:1.9.2
    command:
      - bash
      - -ce
        FQDN=$(hostname -f)
        PROP_FILE=${NIFI_HOME}/conf/nifi.properties
        p_repl () {
            echo "setting ${1}=${2}"
            sed -i -e "s|^${1}=.*$|${1}=${2}|" ${PROP_FILE}
        }
        p_repl nifi.remote.input.host ${FQDN}
        p_repl nifi.cluster.is.node true
        p_repl nifi.cluster.node.protocol.port 6007
        p_repl nifi.cluster.node.address ${FQDN}
        p_repl nifi.cluster.protocol.is.secure false
        p_repl nifi.security.user.authorizer managed-authorizer
        p_repl nifi.web.http.host ${FQDN}
        p_repl nifi.web.http.port 8080
        p_repl nifi.zookeeper.connect.string ${NIFI_ZOOKEEPER_CONNECT_STRING}
Apply the NiFi StatefulSet configuration:

```bash
$ kubectl apply -f 40-statefulset.yml
```

Lastly, create an Ingress configuration for NiFi in a file named 50-ingress.yml from Listing 9-4. Apache NiFi supports authentication, however, this requires it to run in SSL mode and needs additional component configuration to manage certificates and Ingress. In keeping this demonstration concise, the Ingress configuration secures NiFi with Basic Auth credentials stored in the Kubernetes Secret sysop-basic-auth.

---

[28] https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#user_authentication
Listing 9-4. NiFi Ingress

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nifi
  namespace: data
annotations:
  cert-manager.io/cluster-issuer: letsencrypt-production
  nginx.ingress.kubernetes.io/auth-type: basic
  **nginx.ingress.kubernetes.io/auth-secret: sysop-basic-auth**
  nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
spec:
  rules:
  - host: nifi.data.dev3.apk8s.dev
    http:
      paths:
      - backend:
        serviceName: nifi
        servicePort: 8080
        path: /
  tls:
  - hosts:
    - nifi.data.dev3.apk8s.dev
      secretName: data-production-tls

Apply the NiFi Ingress configuration:

$ kubectl apply -f 50-ingress.yml

Verify the new NiFi cluster is up and running by browsing to https://nifi.data.dev3.apk8s.dev/nifi, open the Global Menu in the upper-left corner of the user interface, and select the item Cluster. Review the list of running nodes as shown in Figure 9-6.
The following section demonstrates an example of an ETL/ELT data pipeline utilizing OpenFaaS installed earlier in this chapter, along with Apache Kafka, Elasticsearch, and JupyterLab introduced in previous chapters.

**Example ETL Data Pipeline**

Traditional examples of ETL (extract, transform, load) operations would likely demonstrate the extraction of data collected and stored in Big Data systems such as HDFS or any variety of commercial and open source data lakes and legacy or modern enterprise data management systems. Although the following example demonstrates the extraction of data from Twitter, it should be easy to appreciate the flexibility of NiFi’s wide range of prebuilt data processors and apply these same essential abilities to nearly any variety of ETL challenges.

This example requires OpenFaaS and the SentimentAnalysis Function installed earlier in this chapter; Apache Kafka, configured in Chapter 5; and Elasticsearch, Kibana, Keycloak, and JupyterHub, introduced in Chapter 6.
The following example ETL data pipeline extracts messages from Twitter with the NiFi Twitter processor and publishes them to Apache Kafka topic. Subsequently, a Kafka processor consumes messages in the topic, preparing and sending them to the OpenFaaS SentimentAnalysis Function, finally storing the results in an Elasticsearch index for analysis within a JupyterLab environment. This example demonstrates the ease in which Kubernetes manages all the required workloads in a distributed, highly available, monitored, and unified control plane (see Figures 9-1 and 9-5).

**NiFi Template**

Apache NiFi provides detailed documentation for users, administrators, and developers looking to extend its capabilities. Covering all the capabilities of Apache NiFi would alone fill many chapters if not the entire book; therefore, to get a quick demonstration of its use, along with utilizing the components configured and installed in this book, a prebuilt template is available at [https://github.com/apk8s/nifi-demo](https://github.com/apk8s/nifi-demo).

Clone the apk8s/nifi-demo repository:

```bash
git clone git@github.com:apk8s/nifi-demo.git
```

After browsing to the running NiFi cluster at [https://nifi.data.dev3.apk8s.dev/nifi](https://nifi.data.dev3.apk8s.dev/nifi), click the template upload button shown in the Operate Palette on the left-hand side of the screen (see Figure 9-7). When prompted, upload the file Twitter_Sentiment.xml found in the templates directory of the apk8s/nifi-demo repository.

---

29 [https://nifi.apache.org/docs.html](https://nifi.apache.org/docs.html)
After uploading the template, drag the template icon (three connected boxes) from the Components Toolbar (top navigation), into the canvas (grid) as shown in Figure 9-8. Before completing, the template component prompts the user with a list of available templates; choose “Twitter Sentiment v2” and click the add button.

Figure 9-7. Apache NiFi upload template

After uploading the template, drag the template icon (three connected boxes) from the Components Toolbar (top navigation), into the canvas (grid) as shown in Figure 9-8. Before completing, the template component prompts the user with a list of available templates; choose “Twitter Sentiment v2” and click the add button.

Figure 9-8. Apache NiFi add template
After adding the template, the canvas now contains ten NiFi processors, as shown earlier in Figure 9-5. The processors provided by the template are preconfigured to utilize components installed in previous chapters, such as Apache Kafka and Elasticsearch. Double-click any processor and select the Properties tab to view its configuration.

Before activating the new dataflow, the processor GetTwitter requires a Consumer Key, Consumer Secret, Access Token, and Access Token Secret provided by Twitter. Generate these values by creating a Twitter account, visiting the portal, and then selecting Apps from the drop-down navigation.30 On the Apps page for Twitter developers,31 click the Create an app button and complete the required steps. Once Twitter approves the new app, retrieve the tokens, keys, and secrets shown in Figure 9-9.

---

30 https://developer.twitter.com/en
31 https://developer.twitter.com/en/apps
Figure 9-9. *Apache NiFi Twitter keys and tokens*

Populate the values required by the *GetTwitter* processor shown in Figure 9-10.
The new dataflow is ready to run. However, Elasticsearch is the final endpoint and requires an index template to store the data fields suitably. The next section adds an index template to Elasticsearch.

**Prepare Elasticsearch**

The NiFi processor `PutElasticsearchHttp` provided by the template puts the final processed data into an Elasticsearch index, matching the pattern `sentiment-${now():format('yyyy-MM')}`, creating a new index for each month of the year. `PutElasticsearchHttp` receives and puts JSON data assembled by the previous processors. This JSON data structure contains text, numeric, and date values. Elasticsearch can detect and automatically set data types, but it’s not perfect and easily confused by varied date formats. Elasticsearch is naturally unable to determine if values such as the number zero are an integer or a float. Enforcing proper indexing is accomplished by providing Elasticsearch with an index template.  

---

**Figure 9-10. Configure the Apache NiFi GetTwitter processor**

The new dataflow is ready to run. However, Elasticsearch is the final endpoint and requires an index template to store the data fields suitably. The next section adds an index template to Elasticsearch.

**Prepare Elasticsearch**

The NiFi processor `PutElasticsearchHttp` provided by the template puts the final processed data into an Elasticsearch index, matching the pattern `sentiment-${now():format('yyyy-MM')}`, creating a new index for each month of the year. `PutElasticsearchHttp` receives and puts JSON data assembled by the previous processors. This JSON data structure contains text, numeric, and date values. Elasticsearch can detect and automatically set data types, but it’s not perfect and easily confused by varied date formats. Elasticsearch is naturally unable to determine if values such as the number zero are an integer or a float. Enforcing proper indexing is accomplished by providing Elasticsearch with an index template.  

---

32 [www.elastic.co/guide/en/elasticsearch/reference/current/indices-templates.html](http://www.elastic.co/guide/en/elasticsearch/reference/current/indices-templates.html)
Index templates consist of a JSON-based configuration, defining how one or more fields should be indexed. The following commands port-forward Elasticsearch and post an index template matching the data types resulting from the processed data.

Open a terminal and port-forward Elasticsearch:

```bash
$ kubectl port-forward elasticsearch-0 9200:9200 -n data
```

Open another terminal and post the index template by issuing the command in Listing 9-5.

**Listing 9-5.** HTTP post an Elasticsearch index template

```bash
cat <<EOF | curl -X POST \
-H "Content-Type: application/json" \
-d @- http://localhost:9200/_template/all
{
  "index_patterns": ["sentiment-*"],
  "settings": {
    "number_of_shards": 1
  },
  "mappings": {
    "_source": {
      "enabled": true
    },
    "properties": {
      "polarity": {
        "type": "float"
      },
      "subjectivity": {
        "type": "float"
      }
    }
}
```

Chapter 9  routing and transformation
Elasticsearch is now able to properly index processed data from the dataflow defined by the NiFi template loaded in the previous section. The following section starts up the dataflow and queries the resulting processed data.
Dataflow

The example ETL data pipeline loaded as a template earlier in this chapter extracts data from Twitter and publishes it to an Apache Kafka topic. Another set of processors consumes data from the Kafka topic, obtains the text of the Twitter message, and sends it to the OpenFaaS Sentiment Analysis function. A final set of processors combines the results of Sentiment Analysis along with fields from the original data and posts the results as JSON to Elasticsearch for indexing.

Twitter produces a high-velocity, endless flow of semi-structured data representing a typical data processing scenario. The use of Apache Kafka in this example is not necessary and only used to demonstrate additional NiFi processors. However, the use of Kafka allows external systems the opportunity to act on its data event stream, providing more opportunity for expanding the pipeline.

To start the new dataflow, click anywhere on the NiFi canvas (grid) and start all data processors by clicking the play button provided by the Operate Palette.

After a few minutes, open a terminal and port-forward Elasticsearch:

```bash
$ kubectl port-forward elasticsearch-0 9200:9200 -n data
```

Open another terminal and post an Elasticsearch query by issuing the command in Listing 9-6. The following query aggregates the last hour of the polarity metric from Sentiment Analysis into histogram buckets at every 0.5 interval from -1 to 1. Elasticsearch supports a robust set of aggregation capabilities.33

33 [www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html](http://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html)
Listing 9-6. HTTP post an Elasticsearch Sentiment Analysis query

cat <<EOF | curl -X POST \
-H "Content-Type: application/json" \ 
-d @- http://localhost:9200/sentiment-*/_search
{
    "size": 0,
    "aggs": {
        "polarity": {
            "histogram": {
                "field": "polarity",
                "interval": 0.5,
                "extended_bounds": {
                    "min": -1,
                    "max": 1
                }
            }
        },
        ...
    }
},
"query": {
    "range": {
        "Date": {
            "gt": "now-1h"
        }
    }
}
EOF
The example results (see Listing 9-7) show there were ten times more negative Twitter posts ("doc_count": 40) regarding COVID-19 than positive ("doc_count": 4) in the last hour.

**Listing 9-7.** Example aggregation output from Elasticsearch

Sentiment Analysis query

```json
{
    "took": 5,
    "timed_out": false,
    "_shards": {
        "total": 1,
        "successful": 1,
        "skipped": 0,
        "failed": 0
    },
    "hits": {
        "total": {
            "value": 2276,
            "relation": "eq"
        },
        "max_score": null,
        "hits": []
    },
    "aggregations": {
        "polarity": {
            "buckets": [
                {
                    "key": -1,
                    "doc_count": 40
                },
```
The example ETL dataflow attempts to demonstrate a small set of the many features provided by Apache NiFi, along with Kubernetes’s ability to provide an ideal platform for the near-seamless interconnectivity of data management, storage, and processing systems. The platform in this book demonstrates Kubernetes’s handling of widely diverse applications, from large monoliths such as NiFi, Elasticsearch, and Kafka to Serverless Functions, wrapped in containers, deployed, monitored, and managed across multiple servers with a unified network and control plane.

The next section leverages a JupyterLab environment, demonstrating the ability for real-time experimentation and interaction with platform data.
Analysis and Programmatic Control

JupyterHub, installed and configured in Chapter 6, provides JupyterLab environments, facilitating the operation of one or more Jupyter Notebooks running directly in the cluster. The following two exercises demonstrate both the simple query and visualization of data indexed in Elasticsearch and the ability to develop NiFi dataflows programmatically.

Analysis and Visualization

This example uses a Python-based Jupyter Notebook provided by JupyterHub (see Chapter 6). As data flow into Elasticsearch, it is immediately indexed and searchable by all its fields. The example returns up to 10,000 records from Elasticsearch indexes starting with sentiment- and the Date field value is within the last hour.

Open a new Python-based Jupyter Notebook and add each of the following code blocks to individual cells.

Install the Elasticsearch package version 7.6.0 by adding the following command to the first cell:

!pip install elasticsearch==7.6.0

Import elasticsearch, pandas, and matplotlib:

```
from elasticsearch import Elasticsearch
import pandas as pd
from matplotlib import pyplot
```

Create an Elasticsearch client connected to the elasticsearch service running in the Kubernetes Namespace data:

```
es = Elasticsearch(["elasticsearch.data"])
```
Use the Elasticsearch client’s search function to query the index pattern sentiment-*, and store the results in the variable response:

```python
response = es.search(
    index="sentiment-*",
    body={
        "size": 10000,
        "query": {
            "range": {
                "Date": {
                    "gt": "now-1h"
                }
            }
        },
        "_source": ["Date", "polarity", "subjectivity"]
    }
)

Map and transpose the response from Elasticsearch into Pandas DataFrame:

```python
df = pd.concat(map(pd.DataFrame.from_dict,
    response['hits']['hits']),
    axis=1)["_source"].T
```

Convert the Date column to a Python Datetime data type:

```python
datefmt = '%a, %d %b %Y %H:%M:%S GMT'
df['Date'] = pd.to_datetime(df['Date'], format=datefmt)
```
Assign the Date field to the DataFrame index and convert all numeric values to floats:

```python
df = df.set_index(['Date'])
df = df.astype(float)
```

Print the first five records (as shown in Figure 9-11):

df.head()

![Figure 9-11. Sample Sentiment Analysis DataFrame rows](image)

Finally, plot sentiment by calling the plot function of the DataFrame, assigning polarity to the y axis (see Figure 9-12):

```python
df.plot(y=['polarity'], figsize=(13,5))
```
The previous example is a rudimentary sample of data analysis and visualization. A data scientist or analyst’s first step may include similar tasks to form a cursory understanding of the available data. Data Science activities such as Machine Learning typically require sets of immutable/fixed data to facilitate reproducible experiments. The ability to connect in-cluster Jupyter Notebooks with MinIO object storage (installed in Chapter 7), along with event queues, data management, and ETL systems, provides many opportunities for efficiently building and sharing these valuable data sets.

The Kubernetes-backed JupyterLab environment provides a suitable platform for interactive programmatic control over cluster resources with internally exposed APIs, such as Apache NiFi; the next section covers a quick example of this.

Figure 9-12. Sentiment Analysis DataFrame plot
Programming NiFi

Apache NiFi supports extension through custom controllers and processors written in Java. However, NiFi’s robust set of standard processors means that many projects will find a set that works for a surprising number of circumstances. Another extension of NiFi’s capability is through the API, facilitating automation and monitoring. This section contains a brief example of creating a NiFi process group and populating it with a single processor.

Open a new Python-based Jupyter Notebook and add each of the following code blocks to individual cells.

Install the NiPyApi Python package version 1.14.3 by adding the following command to the first cell:

```
!pip install nipyapi==0.14.3
```

Import the package:

```
import nipyapi
```

Configure the NiFi client with the API endpoint, in this case the Kubernetes Service `nifi` in the data Namespace:

```
api_url = "http://nifi.data:8080/nifi-api"
nipyapi.utils.set_endpoint(api_url)
```

---

34 [https://medium.com/hashmapinc/creating-custom-processors-and-controllers-in-apache-nifi-e14148740ea](https://medium.com/hashmapinc/creating-custom-processors-and-controllers-in-apache-nifi-e14148740ea)

35 [https://nifi.apache.org/docs/nifi-docs/rest-api/index.html](https://nifi.apache.org/docs/nifi-docs/rest-api/index.html)

36 [https://github.com/Chaffelson/nipyapi](https://github.com/Chaffelson/nipyapi)
Test the client connection by retrieving information on the first node in the cluster; Figure 9-13 depicts example output:

```python
nodes = nipyapi.system.get_cluster().cluster.nodes
nodes[0]
```

![Image of NiFi Python client node output]

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```

Figure 9-13. NiFi Python client node output

Create a NiFi Process Group and place it on the canvas (above the processors added earlier in this chapter):

```python
pg0id = nipyapi.canvas.get_process_group(
    nipyapi.canvas.get_root_pg_id(),
    'id'
)
```
After executing the cell with the code in the preceding example, visit the NiFi web interface and note the new `apk8s_process_group_0` process group above the processors added earlier as shown in Figure 9-14. Double-clicking the new process group reveals a blank canvas.

![NiFi Process Group added by the Python API client](image-url)

**Figure 9-14.** NiFi Process Group added by the Python API client
Create a GenerateFlowFile\textsuperscript{38} processor in the new NiFi Process Group created previously and place it on the canvas:

\begin{verbatim}
gf = nipyapi.canvas.get_processor_type('GenerateFlowFile')

pg0 = nipyapi.canvas.create_processor(
    parent_pg=pg0,
    processor=gf,
    location=(250.0, 0.0),
    name="apk8s_processor_0",
    config=nipyapi.nifi.ProcessorConfigDTO(
        scheduling_period='1s',
        auto_terminated_relationships=['success']
    )
)
\end{verbatim}

After executing the cell with the code in the preceding example, visit the NiFi web interface, double-click the new process group, and view the newly created GenerateFlowFile as shown in Figure 9-15.

\footnote{https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-standard-nar/1.11.4/org.apache.nifi.processors.standard.GenerateFlowFile/index.html}
This section automates the creation of a NiFi Process Group along with a NiFi GenerateFlowFile Processor. This example demonstrates a granular method of dataflow development. The NiFi API and NiPyApi Python package also support the installation and configuration of templates, allowing developers to design and save a variety of complete dataflows as templates, such as the Twitter Sentiment v2 (added earlier in the chapter), making them available for programmatic configuration, deployment, and monitoring.

Figure 9-15. NiFi Processor added by Python API client
Summary

This chapter installed the Serverless platform OpenFaaS and the data routing and transformation platform Apache NiFi (see Listing 9-8), demonstrating interconnectivity with other data management components installed from previous chapters, specifically Apache Kafka, Elasticsearch, and JupyterLab. This chapter showed the extraction, transformation, loading, processing, and analysis of Twitter messages all without the need for custom code, yet provides many ways to extend with code, from writing custom (Serverless) Functions for OpenFaaS or interacting with Elasticsearch, Kafka, and NiFi interactively through Python in JupyterLab.

This book aims to demonstrate the ease in which a data platform may be quickly assembled, managed, and monitored atop Kubernetes. The level of integration with Kubernetes is wide ranging. Software such as OpenFaaS and JupyterHub utilizes the Kubernetes API themselves for deploying and scaling Pods, while others like NiFi operate with no awareness of Kubernetes.

The Kubernetes data platform developed in this book runs on a small-scale, resource-constrained, four-node development cluster, costing only a few dollars a day. Yet, this small cluster covers many fundamental data handling concepts, including data events, indexing, processing, databases, data lakes, data warehouses, distributed query execution, modern ETL operations, and data science environments. These capabilities are essential for organizations expected to collect, process, and analyze a variety of data. IoT and machine learning are examples of concepts with heavy demands on data management ranging from the collection of high-velocity real-time unstructured and semi-structured data to processed, normalized, well-structured data catalogs for training and refining machine learning models.
Listing 9-8. Organization of Kubernetes-based data platform components

./009-cluster-apk8s-dev5
    ├── 000-cluster
    │    └── 003-data
    │        ├── 000-namespace
    │        │    └── 005-keycloak
    │        │    ├── 010-zookeeper
    │        │    ├── 020-kafka
    │        │    ├── 030-elasticsearch
    │        │    ├── 032-logstash
    │        │    ├── 034-kibana
    │        │    └── 050-mqtt
    │        │            ├── 060-cassandra
    │        │            ├── 070-minio
    │        │            ├── 080-mysql
    │        │            ├── 085-hive
    │        │            ├── 095-presto
    │        │            ├── 100-jupyterhub
    │        │            └── **120-openfaas**
    │        │                    └── **150-nifi**
    │        └── 005-data-lab
    │            └── 000-namespace