CIREI

Data Analytics with WISE-WARE

Contents

CIREI

Introduction

Important concepts

Data production

Smart home devices produce data; motion sensor events, thermometer readings, a window being opened or closed, a light being toggled or some other command being initiated by the user. In WISE-WARE, this data is produced to Apache Kafka as a stream of events, kept separate from the service that produces them.

Data consumption

To manipulate or generate insights from data, it needs to be consumed. This does not destroy the data as it exists in the message queue, but the consumer tracks which event it last read. In the event of failure, it can continue where it stopped without consuming duplicate data.

Message queues and decoupling

Making data production and consumption separate tasks performed by separate entities is called “decoupling”. The production and consumption of data do not directly impact each other but do so through an intermediary “message queue”; Apache Kafka performs this role in stock WISE-WARE. If a producer or consumer has problems or is overloaded, the message queue acts as a buffer to allow the other to continue as normal.

Connector modules

Because there are many different possible data producers and consumers - whether they are other smart home solutions or visualisation suites for example - there is a need to move data between them effectively. The components that do that are “connectors”, and are assumed to be sufficiently developed that they allow complete conversion from 1 system’s format to another. These connectors are often distinguished by whether they pull data into or out of the reference system (Kafka, in the case of stock WISE-WARE), referred to as “source” connectors in the former case and “sink” connectors in the latter case.

Data transformations

In addition to conversion, connectors may also be configured to transform data as it passes through. This can be as simple as adding a new field with the timestamp of the data point’s consumption, renaming, reorganising or removing existing fields, or completely changing the data’s structure.

Key-value manipulation

Transformations may be performed to assist organisational functions. Kafka events may have both a “key” and a “value”; when a Kafka event is transformed, fields may be moved from the key to the value, or vice-versa, to assist with sorting or filtering operations further down the data stream.

Real-time and batch processing

Data processing always occurs over some length of time; this is partially a product of the limitations of the hardware and software used, but can also be a result of intentional system design decisions. The 2 contrasting extremes of this principle are “real-time processing” and “batch processing”. The choice to include one or the other (or both) depends on the nature of the data and the desired analysis outcomes.

Real-time processing takes place continuously, using data that is as current and timely as possible - note however that “now” in any distributed system is an illusion (Justin Sheehy wrote an approachable article exploring this; https://queue.acm.org/detail.cfm?id=2745385). This is often done in cases where the data’s insights are time-critical and needed immediately after creation. For example: alerting caregivers of a possible fall based on sensor readings.

Due to the limited lifespan of the data being processed - after some amount of time, new and more relevant data will be produced - real-time processing cannot be done on extremely large datasets. The time it would take to complete would at least partially invalidate the insights gained.

In cases where these insights are not time-critical (relative to the time taken to generate) a pipeline might instead wait for a certain volume of data to be produced - perhaps over 24 hours - and then process that data all at once. This approach is referred to as batch processing, and is typically used to generate reports from historical data with more complex and detailed analysis than would be possible in real time. It allows larger datasets to be utilised and fully explored, guiding decisions that have effects propagating further into the future.

Both real-time and batch processing can be used together in a single business intelligence solution, as the data pipeline diverges and the same source data is put towards different uses.

Kafka software ecosystem

“Apache Kafka” refers not just to the brokers themselves, but also to the supplementary software that is designed to interface with them. The principal elements discussed here are “Kafka Connect” and “Kafka Streams”, although other software with compatibility do exist.

Kafka Connect

Kafka Connect is the main connector software for allowing data to be exchanged between Kafka brokers and other systems, such as databases, message streaming solutions, archival or analysis services, and even other Kafka brokers. Kafka Connectors are developed as self-contained and easily-configurable components, so they can be installed and utilised with minimal knowledge of either end of the transfer. They can be used to move data in a batch transaction during testing and development, or integrated as part of a more permanent or real time pipeline.

In addition to allowing transfer of the raw data, Kafka Connectors can be configured to include “Single Message Transforms” (SMTs). These apply transformations as discussed earlier, and can include filtering and routing operations as well. Filtering operations are used to specify which data are moved, and routing operations specify where they are moved to. These 2 additional features allow for complex data flows to be defined and implemented quickly and with minimal knowledge of programming. This in turn makes Kafka Connect especially useful in prototyping, or when used by team members without extensive Java expertise.

Kafka Connect is primarily limited by the functionalities of the connectors and transformations used. While both can be found online in abundance, and can be configured and combined to great effect, especially demanding data architectures may not have their needs met. In such a case new modules can be developed using Java, possibly to be combined with existing ones.

Kafka Streams

For cases where Kafka Connect does not fulfil the requirements of a data pipeline, new Java applications can be written using Kafka Streams. Kafka Streams is a Java library offering multiple Java interfaces and classes for working with data in Kafka, as well as a high-level domain-specific language (DSL) to lower the barrier to entry.

Streams can be used for too many purposes to be adequately covered here, but a few examples include:

- Consuming data from a single topic, refining or restructuring it in some way, and producing to another topic on the same broker
- Consuming data from a single topic and filtering/masking the events to produce a more restricted and semantically-relevant view of the data
- Consuming data from many topics and aggregating them into a set of representative state variables, which are then produced to another topic for compression purposes or high level overviews in visualisation suites further down the data pipeline

These can be implemented using conventional algorithms written in Java using the library functionalities provided by Streams, but the DSL is an alternative option that provides many options usable declaratively. Similarly to, although still very distinct from, the benefits provided by Kafka Connect, this lowers the required Java expertise to implement complex data flows while also providing more options and control to developers.

External components

WISE-WARE leverages Apache Kafka internally and recommends it as a dominant component in the data fabric of business intelligence solutions, for data movement and processing. However, the power of a business intelligence solution lies in its ability to integrate with and gather data from diverse sources, generate rich insights and deliver them in an effective manner. Therefore, an introduction to commonly-found external components is presented here. Later, 2 practical examples showing integration between WISE-WARE and an existing system will also be presented.

Databases

Databases are the fundamental type of software used for data storage and retrieval. However, different types exist which are optimised for specific purposes. Relational databases, like MySQL or PostgreSQL, excel at storing and managing structured data with well-defined relationships between different entities. These databases are ideal for storing information that doesn't require time-series analysis, such as customer records, product details, or financial transactions.

Time-series databases

For data that has a temporal component, such as sensor readings or stock prices, time-series databases (TSDBs) are more efficient. Unlike relational databases, TSDBs are specifically designed to handle high volumes of time-stamped data points. They excel at writing data quickly and efficiently, even for large data streams, while optimizations may limit functionality for updating or deleting existing data points (since these operations are less common in time-series analysis). InfluxDB is a popular example of a TSDB, and has a connector solution in the form of Telegraf for directly ingesting real-time sensor data or other time-series data streams.

Other message queues

While Kafka excels at handling high-volume data pipelines for complex analytics, other message queue protocols exist that cater to different needs. Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol optimised for real-time communication at a smaller scale. MQTT is ideal for situations where real-time communication between devices is crucial, such as connecting individual smart home devices that publish sensor readings or receive control commands. WISE-WARE can integrate with MQTT through connectors, potentially translating and forwarding relevant data to Kafka for further processing and analysis alongside data from other sources.

Visualisation suites

Once data has been collected, processed, and analysed, it's crucial to present the insights in a way that is easily understandable for users, even those without a technical background. Visualisation suites provide user-friendly interfaces for displaying raw data and insights. They render data into graphs, charts, and other visual formats, allowing users to explore trends, identify patterns, and gain valuable insights from the information. A popular example is Grafana, an open-source platform for creating interactive dashboards and visualising data. Integrating WISE-WARE with visualisation tools allows users to interact with the data directly, facilitating deeper exploration and aiding data-driven decision-making. This may assist care services with monitoring their residents, but could also be used with the residents themselves. Visualisations may assist with framing particular topics in ways that are more agreeable, such as a new treatment or intervention option for a chronic condition.

AI/ML analysis

Data analysis can be performed using various programming languages, tools, and statistical techniques. When working with sufficiently large datasets, machine learning (ML) offers a powerful approach for identifying subtle relationships and extracting deeper meaning from the data. ML algorithms can learn from historical data and use that knowledge to make predictions, identify anomalies, or classify data points into specific categories.

One of the most popular frameworks for building and deploying ML models is TensorFlow. For more information, consider TensorFlow's official material on integration with Kafka: https://www.tensorflow.org/io/tutorials/kafka. By integrating with ML tools like those made accessible by the likes of TensorFlow, extensions on stock WISE-WARE uncover hidden patterns and gain deeper insights from the data. These insights can aid in tasks like predictive maintenance, anomaly detection, or risk assessment.

Examples and guides for integrating WISE-WARE with external systems

These sections walk through the basic steps of setting up a Kafka Connector, which can be reapplied and configured to ingest data from or export data to various sources, and a Kafka Streams application, which can be used to permute data in ways more advanced than is conventionally possible using just single message transformations in a connector. The code examples will use real code from the case study that took place in the University of Nottingham (UoN) Cobot Makers Space (CMS) to illustrate the process of integration.

Connect

For a given connector configuration, there are at least 2 configuration files that are needed. The first configures the Connect worker, a process that actually carries out the data transfer and transformation. The second configures the connector itself, which is a module contained within and utilised by the connect worker.

A last consideration is that Kafka Connect can operate in 1 of 2 modes; standalone and distributed. For deployment purposes, distributed mode is preferable due to it's ability to improve fault tolerance and processing capacity through horizontal scaling, and so it will be used in these examples.

In distributed mode, communication with the Connect cluster takes place using a RESTful API through http requests, so a tool capable of creating and sending these requests is required. These examples will use curl, a utility executable bundled with most operating systems.

Configuration

The worker configuration defines the overall settings for the Connect worker itself, in particular:

- Bootstrap servers: the list of addresses (and ports) of the Kafka brokers in the cluster that the Connect worker will try to communicate with
- Group id: a unique identifier for the cluster, is used to allow multiple instances of the Connect worker to cooperate on tasks; in cases where there is already a consumer cluster, care must be taken to ensure that this group id does not conflict with it
- Key and value converters, and schema settings: Kafka records have botha a key and a value, but the format in which they are stored can vary widely. Sometimes they are stored in plaintext or as bytearrays, while other times they will be Json-formatted and with a particular schema. The serialiser and deserialiser used can be different for the key and value if needed, and a specific schema can be defined and included if it is relevant
- Replication factors: in deployments featuring many kafka brokers cooperating as a cluster, data can be replicated multiple times for durability. The number of replicas is configurable, and can be configured differently for the various supporting data created and used by the Kafka Connect workers themselves (which are also stored in the Kafka broker they operate on)

- Plugin path: by default, Kafka Connect workers have access to a small set of connector plugins, so additional ones need to be downloaded and installed. The file path to the directory with these connectors needs to be defined (and the worker restarted, if new plugins have been added since it started running)

An example of a configuration file for a Connect worker:

```
# A list of host/port pairs to use for establishing the initial
connection to the Kafka cluster.
bootstrap.servers=wiseware:9092

# unique name for the cluster, used in forming the Connect cluster
group. Note that this must not conflict with consumer group IDs
group.id=connect-cluster

key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.converters.ByteArrayConverter

key.converter.schemas.enable=false
value.converter.schemas.enable=false

offset.storage.topic=connect-offsets
offset.storage.replication.factor=1

config.storage.topic=connect-configs
config.storage.replication.factor=1

status.storage.topic=connect-status
status.storage.replication.factor=1

offset.flush.interval.ms=10000

plugin.path=/home/cirei/Documents/kafka_2.13-3.6.1/libs
```

After starting the connect worker, you need to configure a connector.

- Name: is used to identify the connector once it has been installed in the worker; needs to be distinct from the worker's other connectors.
- Connector class: the Java class for the connector plugin; needs to have been included in the configured plugin path. Each connector plugin has it's own configuration options which will also need to be included in addition to the general options discussed here.
- Maximum tasks: in cases where there are many topic partitions on which to operate, a maximum number of tasks can be defined that the worker instantiates. The amount of parallelism that can be achieved in practice is highly dependent on the specific connector plugin used and the composition of the clusters of Kafka brokers and Kafka Connect workers.

- Key and value converter overrides: optional parameters that can override the key and value converters that the worker is configured to use, in cases where there are many connectors that do not all need the same converter settings.

An example of a connector configuration file, used at UoN CMS to ingest data from their MQTT broker:

```
{
    "name": "mqtt-source",
    "config": {
        "connector.class":
"io.confluent.connect.mqtt.MqttSourceConnector",
        "tasks.max": 1,
        "kafka.topic": "mqtt",
        "confluent.topic.bootstrap.servers": "172.25.255.222:9092",
        "confluent.topic.replication.factor": 1
        "mqtt.ssl.key.store.path":
"/home/cirei/Documents/kafka_2.13-3.6.1/cireiuonconfig/uonmqtt/truststor
e.jks",
        "mqtt.ssl.key.store.password": "wisewarekeystorepassword",
        "mqtt.server.uri": "ssl://172.24.10.12:1883",
        "mqtt.topics": "instrumentation/+/state",
    }
}
```

In the above case, in addition to configuring access to the MQTT broker the confluent bootstrap server and replication factors are defined. Because this connector uses a plugin made by Confluent, it's licence details must be stored somewhere (and this does not always need to be on the same brokers that the business-relevant data is being read or written).

Connectors will have their own variant of “kafka.topic” or “kafka.topics” or similar to define which topic(s) they push or pull data to or from, depending on if they are source or sink -type connectors.

Data Transformations

You can optionally define transformations to be applied to the data during transfer. These transformations are specified within the connector configuration using specific classes and properties. Some transformations might require predicates, which are conditions used to selectively apply the transformation.

The Confluent documentation explores the options and methods for doing this, and as transformations are highly context-specific they are deemed beyond the scope of this document.

Interacting with the Connect Worker

As earlier-mentioned, the Connect Worker can be interacted with via a RESTful API using HTTP requests. Assuming the earlier steps have been taken to make configuration files, start the Connect Worker from the directory where Kafka is installed:

```
./bin/connect-distributed.sh ./config/<config-filename>.properties
```

The following requests can be used to perform common tasks, again from the same directory:

Command	Function
curl http://localhost:8083	Returns some worker metadata, confirming that it is running as expected.
curl http://localhost:8083/connectors	Queries the connectors currently installed to the worker.
curl http://localhost:8083/connector-plugins	Queries the connector plugins currently available to the worker, can be used to confirm that new plugins are correctly installed
curl -X PUT http://localhost:8083/connectors/<connector-name>/stop && curl -X DELETE http://localhost:8083/connectors/<connector-name>/offsets && curl -X PUT http://localhost:8083/connectors/<connector-name>/connectors/<connector-name>/resume	Stops a given connector from being run, deletes it's offsets, and restarts it. This can be used to reset the progress a given connector has made through it's source data, essentially allowing it to start processing from the beginning of it's data again. Without doing this the connector would retain it's current position, so if it's settings were changed (such as adding a message transformation) the changes would only take effect on new events.
curl -d @./<connector-config>.json -H "Content-Type: application/json" -X POST http://localhost:8083/connectors	Uploads a connector configuration to the worker, instantiating a new connector that immediately starts consuming or producing data.

Streams

The code examples here will use the example of parsing Json-format data which was written to Kafka by Home Assistant. This data will have all but 3 of it's properties removed - "state", "entity_id" and "last_changed" - which will be written to another Kafka topic; this would be part of a larger data pipeline that does not need the rest of the original message so discards it to save bandwidth.

Development

The Kafka documentation has an easy-to-follow quickstart guide for creating streaming applications using Kafka Streams. It requires that Java 8+ and Maven are installed before starting. Assuming that stock WISE-WARE has been configured, following this guide using either the Kafka broker in WISE-WARE or a new broker on a development machine will allow you to start processing data from a deployment quickly.

As stock WISE-WARE features Home Assistant as the main data source which writes to Kafka, an application that can extract and operate on that data was written during the visit to UoN CMS. This data is produced in Json format, so an additional library - Gson - was used for parsing. The Java class comprising this Streams application is in an appendix at the end of this document.

Useful commands for testing Streams applications

To recompile the project, such as after editing files or adding dependencies, run:

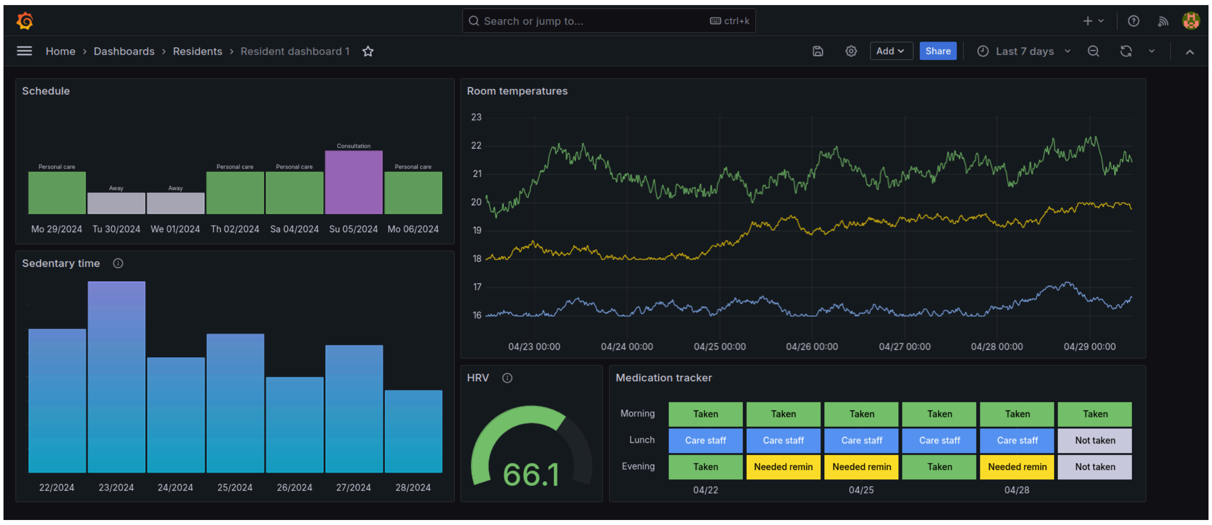
```
mvn clean package
```

To start a Streams application:

```
mvn exec:java -Dexec.mainClass=<package-name>.<streams-app-name>
```

Visualisation

After using Kafka Streams to process, filter and manipulate data as needed, it can be visualised. One such solution for this purpose is Grafana, which can be used to create a dashboard containing real time data, historical records, and referenceable information. Consider the following example which could be part of a care service's tools for monitoring and assisting in the care of their patients:



Morning	Taken	Taken	Taken	Taken	Taken	Taken
Lunch	Care staff	Care staff	Care staff	Care staff	Care staff	Not taken
Evening	Taken	Needed remin	Needed remin	Taken	Needed remin	Not taken
	04/22		04/25		04/28	

Appendix - Kafka Streams example application

The following Java class parses Json-format data written to a Kafka broker by Home Assistant, then removes unwanted properties. The output is re-written to another Kafka cluster, and is of less variable structure

```
package myapps;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;

import java.util.Arrays;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

public class TruncatePipe {
   public static void main(String[] args) {
       //Properties from the input data to remove
       //The source data has a known structure containing 6 properties,
3 of which are irrelevant; attributes and context in particular are
objects of variable internal structure, so are harder to process
without more thorough preprocessing and filtration
       //For monitoring purposes, entity_id, state and last_updated are
the most important.  last_updated and last_changed are distinct; the
latter only changes in response to state changes, while the former also
changes when attributes of an entity change
       final String[] remove_list = new String[] {
           //“entity_id”,
           //“state”,
           "attributes",
           "last_changed",
           //"last_updated",
           "context"
       };

       Properties props = new Properties();
       props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
```

```
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        //Defining of the streaming application
        final StreamsBuilder builder = new StreamsBuilder();

        builder.<String,String>stream("home_assistant").flatMapValues(
            value -> {
                JsonObject json =
JsonParser.parseString(value).getAsJsonObject();

                for(String s: remove_list){
                    if (json.has(s)){
                        json.remove(s);
                    }
                }

                return Arrays.asList(json.toString());
            }
        ).to("ha_cleaned");

        //Finalising the stream application; it can now be started
        final Topology topology = builder.build();
        final KafkaStreams streams = new KafkaStreams(topology, props);
        final CountDownLatch latch = new CountDownLatch(1);

        //Attach shutdown handler to catch control-c
        Runtime.getRuntime().addShutdownHook(new
Thread("streams-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```