Implementing an Open Data ETL Processing Engine with Kafka

MASTER THESIS

Fabian Arnold

Submitted on 1 April 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Georg Schwarz, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 1 April 2022

License

Erlangen, 1 April 2022

Abstract

The JValue project group is developing a modeling ecosystem for Extract Transform Load (ETL) processes. Part of this ecosystem is a description model for those. This thesis suggests a conversion process from the description model into an Apache Kafka runtime, described in a cloud-native format, like Docker Compose. The conversion is implemented as a library and done in a multi-phase approach as known from classical compilers. In the first step, the description language is converted into a runtime independent intermediate description and afterward in a description of a concrete runtime, in this case, Kafka. The multi-phase approach minimizes the implementation work for additional runtimes and allows runtime independent optimization and analysis. The goal for the generated runtime is to use existing Kafka components, which is only partially possible due to the complexity of the description model.

Contents

List of Figures

List of Tables

Acronyms

ETL Extract Transform Load

ODS Open Data Service

CRD Concrete Representation Description

IRD Intermediate Representation Description

API Application Programming Interface

DAG Directed Acyclic Graph

VFS Virtual File System

JSON JavaScript Object Notation

CSV Comma Separated Values

HTTP HyperText Transfer Protocol

REST Representational State Transfer

FTP File Transfer Protocol

JRE Java Runtime Environment

SQL Structured Query Language

1 Introduction

Apache Kafka is used by 60% of the Fortune 100 companies. Therefore it is one of the most popular open-source stream-processing software('Apache Kafka powerby', n.d.). Kafka allows reading and writing event streams, provides Application Programming Interfaces (APIs) to extract or load data from other systems, and provides APIs for data transformation. The process of data extraction, data transformation, and data loading is often referred to as ETL.

The vision of the JValue[1] project is to provide an open-source ETL pipeline that extracts open data, optimizes it, and makes it available for developers. This thesis introduces a library that converts an ETL process's description into a Kafka system's concrete configuration.

As the domain of this conversion, a newly arising description model for ETL processes, the so-called Kernel, was used. The Kernel is a description model, similar to the class-object model of object-orientated programming languages, developed by the JValue group. Features of the model are high flexibility in the descriptive capabilities and easy extensibility via user-defined packages.

The conversion uses a multi-phase design as it is used for classical software compilers. A process based on the multi-phase design of classical software compilers is used for the conversion. The input description (Kernel) is converted into an intermediate description independent of the final runtime (Kafka). This description allows optimization and analysis of the ETL process. The final phase converts the intermediate description into a description of concrete Kafka components. A design goal was to use existing components for Kafka and a cloud-native design.

An ETL package for the Kernel was created for the conversion implementation, as no existing one was available. The package supports file and HyperText Transfer Protocol (HTTP) sources, parsing of Comma Separated Values (CSV), transformation via JavaScript, and output via HTTP and a Representational State Transfer (REST) web server.

This thesis does not focus on the deployment, installation, and management

[1] https://jvalue.org/

of Kafka Components, only on the conversion process itself. For deployment, installation, and management an already existing solution - Docker Compose - was chosen with the flexibility to add support for other solutions in mind. This flexibility was kept in mind for adding support to other streaming engines.

The final conversion library can load and enrich data in a real-world scenario. This is shown with an example data source containing information of COVID-19 cases as a CSV list published on GitHub. This list is fetched and processed into a well-defined JavaScript Object Notation (JSON) structure and published as a REST endpoint.

Lastly, problems are shown that arise from the high flexibility the Kernel provides and the chosen approach. An alternative approach that could solve those problems is suggested. The alternative one is the usage of components that are specially designed and implemented for the Kernel.

2 Fundamentals

This chapter gives an overview of this thesis's used technologies and concepts. The first section introduces ETL, followed by an overview of the Apache Kafka ecosystem, orchestration in the IT domain, and the JValue Open Data Service (ODS) with its description component, the Kernel, at the end.

2.1 Extract Transform Load

ETL describes the process of data **e**xtraction, **t**ransformation, and **l**oading. An ETL system extracts data from one or multiple source systems. Examples of those systems are databases, files, and the semantic web. The next step in the ETL process is the transformation of the data. Typical transformation tasks are data quality and consistency enforcement and the combination of different data sources. Finally, the data is loaded into one or multiple target systems in a presentation-ready format and can be easily used by application developers or end-users (Kimball & Caserta, 2011, p. xxi).

2.2 Apache Kafka

Apache Kafka implements a publish/subscribe (pub/sub) messaging system. A pub/sub messaging system is where a sender (producer, publisher) publishes messages not explicitly directed to a receiver (consumer, subscriber). The sender classifies its messages so that a receiver can subscribe to specific classes of messages. The messaging system notifies the receiver of new messages in the subscribed classes. An everyday use for pub/sub-system is the field of enterprise application integration (Narkhede et al., 2017, p. 1).

2.2.1 The Basics of Apache Kafka

A message in Kafka consists of an array of bytes and can have an optional key that also consists of bytes (Narkhede et al., 2017, p. 4).

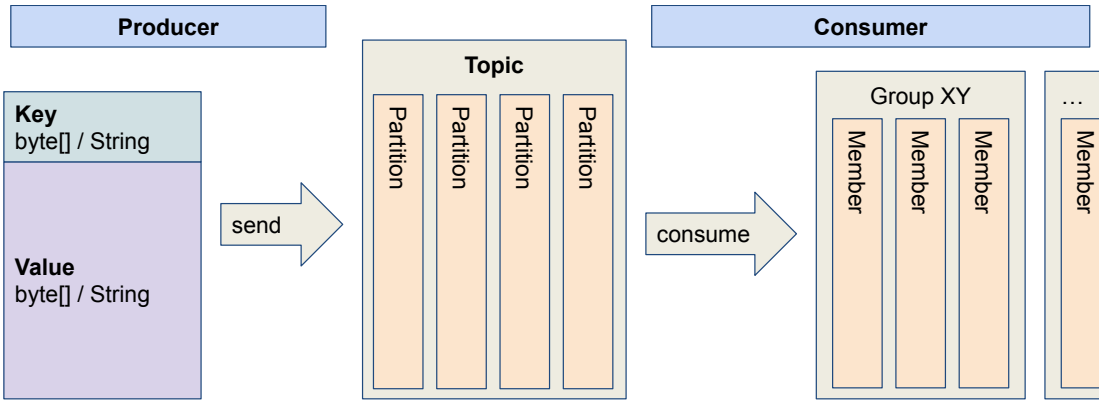


Figure 2.1: The structure of a message, of a topic, and a consumer group in Kafka

Messages in Kafka are classified into topics. A topic is an append-only stream of messages. Every topic has a name and is parted into at least one partition. A partition is the smallest unit of a message stream in Kafka. Messages have a strict order in a partition but not in a stream (topic). The message key is used to sort the messages into a partition. Kafka achieves scaling by sharing partitions across multiple servers (brokers) (Narkhede et al., 2017, p. 5).

The basic clients of Kafka are consumers and producers. A producer creates messages in a topic (Narkhede et al., 2017, p. 43). An example for a producer could be a temperature sensor that reports its measured temperature and location in cyclic intervals into a topic "temps". Consumers are organized into groups and read messages from a topic. Every member of the consumer group gets a fixed set of partitions assigned which messages it consumes (Narkhede et al., 2017, p. 64). For example, one consumer group can read the temperature values and publish them on a website, another could send a notification if the value reaches a specific threshold. A visual representation of this process can be seen in fig. 2.1.

Kafka provides persistent data retention and allows the definition of retention rules per topic. A consumer group that goes offline because of a failure or maintenance will not miss any messages. The retention rules can specify a maximum size or timespan the message is persisted (Narkhede et al., 2017, p. 10). If the limit is reached, Kafka deletes those messages or performs compaction of the topic. Compaction removes messages that share a key and only keeps the latest message with this key available; this allows using Kafka as a data storage.

2.2.2 Kafka Connect

Kafka Connect provides a more sophisticated client and runtime. It covers the extract and load steps of the ETL process and provides developer-friendly APIs

for scalable and failure-tolerant integration of other data sources. The core API consists of two interfaces *Connector* and *Task*. The connector's job is to break down the work into tasks that multiple workers then execute. An example of a task could be "fetch row with id 123 from the database". A worker picks up this task and executes it (Narkhede et al., 2017, pp. 152–153). There are two types of connectors sources and sinks. A source connector is responsible for moving data into Kafka, a sink connector out of Kafka.

2.2.3 Kafka Streams

The Kafka Stream covers the missing transform part of the ETL process. It is a library that allows developers to write applications for processing messages out of Kafka and storing them back into Kafka.

The Streams API allows designing a topology in the form of a Directed Acyclic Graph (DAG) that describes messages' transformation. The topology can contain different processors like filters, mapping, aggregates, and others (Narkhede et al., 2017, pp. 272–273). In contrast to the Kafka Connect library, there is no runtime provided.

2.2.4 Kafka Message Data Format

The Connect and Streams APIs of Kafka allows the developer to define a serializer/deserializer for messages. Internally Kafka stores messages as a byte array, but the format of those messages is often JSON or Avro[1]. A common approach for messages in Kafka is to store the schema definition externally in a schema registry and add a reference to the schema in the first bytes of the message. The serializer and deserializer are aware of those bytes and can look up the schema and validate the messages.

2.3 Mechanics of Software Compilers

A compiler is a piece of software that translates text written in a programming language into a form that can be executed by a computer (Aho et al., 2007, p. 1). Internally it is organized into different phases. The first one starts with a character stream as input. The stream moves through multiple phases till an intermediate representation is generated (front-end of the compiler). The intermediate representation is a platform-independent program description that can be used to apply optimizations. The compilation process continues with the synthesis of the intermediate representation to the target-machine code (back-end) (Aho et al., 2007, pp. 4–5)

[1] https://avro.apache.org/

2.4 Orchestration

In IT, orchestration manages different resources like containers, volumes, and networks. A system that fulfills this task is called an orchestrator. A container is a bundle of software that is isolated. In contrast to a virtual machine, a container shares the host's operating system Kernel, resulting in less resource usage.

A famous container runtime is the Docker Engine[2]. In the case of Docker, a container is created from an image. An image defines the filesystem, environment variables, and startup command of the container. Docker images use a layered design, which means a docker image can extend an already existing image. The layered design simplifies the packaging of applications. A Java application, for example, maybe based on a Java runtime image, which may be based on an image containing a Linux distribution like Ubuntu.

A prominent orchestrator for the Docker Engine is Docker Compose[3]. It takes a description of resources as input and creates, updates, or removes resources based on this file. It supports managing a single node docker installation and cluster configuration based on Docker Swarm[4].

2.5 The Kernel

A topic of the current research of the JValue project group is a generic framework for data representation and data processing. The framework contains multiple components. One of them is a description model for data representation and processing, the so-called Kernel. When this thesis was written, the Kernel was undergoing rapid development and changes, so the following section covers the state of the Kernel in March 2022.

The Kernel defines itself by providing three base objects *MetaType*, *Element*, and *ElementType*. *ElementType* allows defining polymorphism by a field called *extends*. An extended object defines all the properties the object it extends from defines. This concept is equivalent to the concept known in object-orientated programming, where you can extend a class. *Element* defines a relation type, which stores the type an element has. Figure 2.2 shows the UML diagram of *Element* and *ElementType* in the TypeScript implementation of the kernel. An object can/must specify values for properties/relations its type defines. A valid object for the type relation is every object that extends from *ElementType*, including *ElementType* itself. *ElementType* extends *Element*, so an *ElementType* also has an type. The type *ElementType* has is the *MetaType*, which extends

[2]https://docs.docker.com/engine/
[3]https://docs.docker.com/compose/
[4]https://docs.docker.com/engine/swarm/

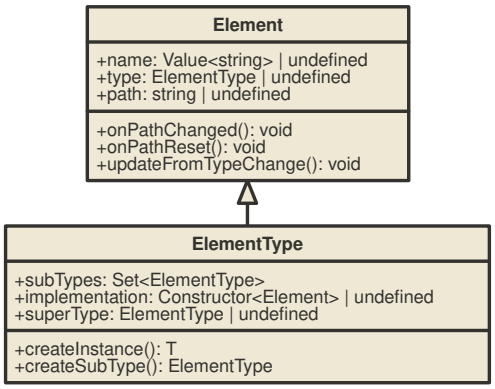


Figure 2.2: UML Diagram of Element and ElementType in the TypeScript implementation of the Kernel.

ElementType. The extension of *ElementType* is needed, as types need to be an *ElementType*. As *MetaType* extends *ElementType*, which extends *Element*, *MetaType* also needs to have a type. *MetaType* now can use itself as its type, as it is an *ElementType*. *Element* is of type *ElementType*, as *ElementType* extends *Element* it also needs a type, which is *ElementType*. These self-referencing definitions of the base elements are the foundation of the Kernel. A visual representation of this can be seen in fig. 2.3.

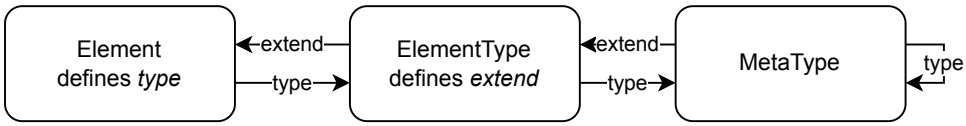


Figure 2.3: Schema of the self-referencing definitions of *MetaType*, *ElementType*, and *Element*

Looking from the perspective of an object-orientated language at the Kernel is not intuitive at the beginning. In an object-orientated language like Java, there is no way to use the *type* relation as in the Kernel. A developer creates a class model based only on the extension of a particular object, like in the case of Java: *java.lang.Object*. The analogy of objects of *ElementType* are classes in Java, exposed via Reflection or the *getClass()* method of an object. At runtime, there are only instances of those objects (classes). In the Kernel model, the developer needs to think of the *type* relation in multiple layers.

To clear up the understanding of the Kernel model, the following contains a simplified graph-orientated description of the Kernel, as the relations *type* and *extends* can be seen as edges of a directed graph. In graph theory, the Kernel

can be considered as the following graph:

$$K = (N, E, T) \tag{2.1}$$

where N are all elements of the Kernel, E are the edges of the extension/inheritance tree (created by the *extends* relation), and T are the edges of the type tree (created by the *type* relation). Every Kernel contains a special element, the so-called *MetaType* M.

$$\{M\} \subseteq N \tag{2.2}$$

The set T, which contains the type tree, can be defined as following:

$$T \subseteq \{(x, y) \mid x, y \in N \text{ and } x \neq y\} \cup \{(M, M)\} \tag{2.3}$$
$$\forall (x, y) \in T \; \exists (x', y') \in T : (y = x') \tag{2.4}$$
$$\forall n \in N \; \exists (x, y) \in E : y = n \tag{2.5}$$
$$\forall (x, y), (x', y') \in E : y = y' \rightarrow x = x' \tag{2.6}$$

eq. (2.3) defines the set T as tuples of elements in N. An entry (x, y) can be read as y is of type x or y is an instance of x. Equation (2.8) enforces that the top-most type is always M. Equation (2.5) enforces that every element of N has a type and eq. (2.6) prevents elements from having multiple types.

The set E which defines the extensions/inheritance of elements can be defined as following:

$$E \subseteq \{(x, y) \mid x, y \in N \text{ and } x \neq y\} \tag{2.7}$$
$$\forall (x, y), (x', y') \in E : y = y' \rightarrow x = x' \tag{2.8}$$

The rules for extensions are less strict. There is no topmost element every element must extend from, and there can be elements that extend nothing. Although it is not enforced, all elements should extend from *Element*, from a semantic perspective.

Based on this definition of the Kernel, we can define additional properties. One of those properties is the model level a node is located in: meta-language, language, model, and instances. Every time an instance of an element is created, the node steps down in its level. Table 2.1 shows all model levels and the distance the nodes have to the *MetaType*. The instance-level is the lowest level, as a type must be of type *ElementType*, which are objects located in the model-level.

Compared to an object-orientated programming language, the programming language works on the model-level. The developer describes and creates its application only at this level. At runtime, instances are created of the developer's model objects (classes).

Knowledge of the level of element in the Kernel is helpful for troubleshooting during development as most converted elements are located on the model layer.

Level	$d(n, M)$
meta-language	0
language	1
model	2
instance	3

Table 2.1: Model-level of Kernel elements based on the distance to the *MetaType*, given by the *type* relation.

Currently, a reference implementation of the Kernel exists in the form of a Typescript library. The reference implementation allows the import and export of the Kernel in JSON format.

The reference implementation of the Kernel also provides a Kernel package, the so-called *system*-package with the following build-in elements:

- **MetaType**: The topmost type in the type tree. Its type is itself. It extends the ElementType.
- **ElementType**: The type a type has. Itself is of type *MetaType*. It extends the *Element*.
- **Element**: A Kernel element; provides basic properties of an element as a path it is located in and a name. Every element that can be added to a *Package* needs to extend from Element.
- **Package**: A Kernel package; provides a list of child elements and packages. Allows the organization and storage of other elements and packages.
- **ValueType**: The type a value has; contains a schema of the value and a list of child value types. It extends from an element.
- **(Int|String|...)Value**: Represents a concrete value, is of type ValueType. Defines a concrete schema of the value it defines and does not extend from an element.

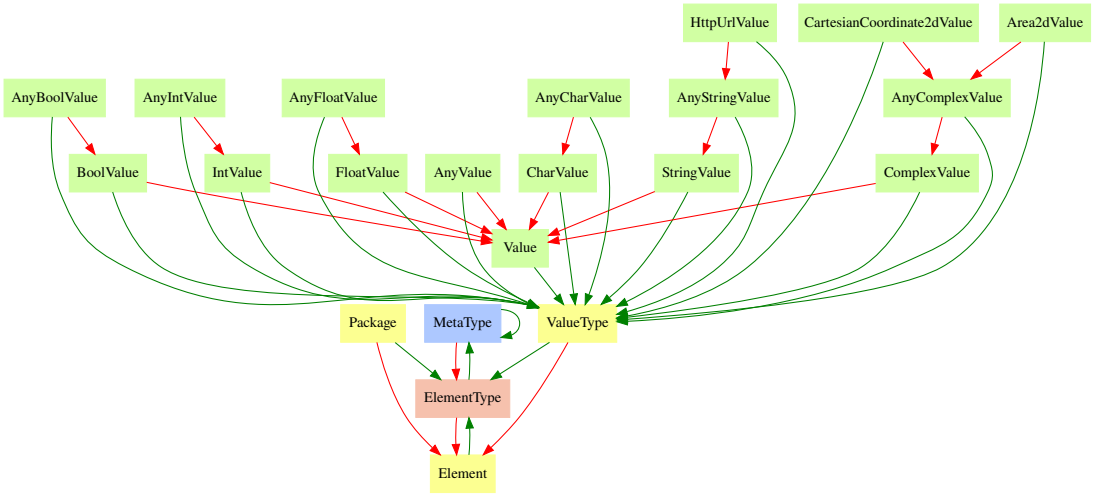


Figure 2.4: Overview of the Kernel elements of the system package, colored by their level: blue is meta-language, red is language, yellow is model, and green refers to instance level. A red arrow specifies the extends relation a green the type relation.

3 Requirements

This thesis provides a library that converts a Kernel from the ODS into a runnable configuration of components related to the Kafka ecosystem. As there are many ways to achieve a conversion, some requirements were defined to specify goals and non-goals of the library. Those requirements can be categorized into non-functional and functional requirements.

3.1 Non Functional Requirements

A non-functional requirement affects the whole architecture of the system and constrains the development of its components(Roman, 1985).

3.1.1 Usage of existing Kafka Components

One of those requirements is the usage of existing Kafka Components. The most straightforward approach to execute the Kernel with Kafka would be the usage of Kafka as a simple database. The application would implement a consumer and producer and handle all the work internally. The problem with this approach is that many benefits that the Kafka ecosystem provides are not used. This approach would lose the main benefit of Kafka Connect and Kafka Streams' scalability. Another problem will be the high implementation and maintenance work required if all logic is implemented from scratch. One design goal was to use existing established components available in the Kafka ecosystem to address the mentioned problems.

3.1.2 Extensibility to Other Runtimes

The Kernel is designed to be a universal description of ETL processes. Kafka may only be one runtime that is used to implement the Kernel. To keep the work needed to add support for other runtimes low, the library should be designed so that adding other runtimes is easy. Possible future runtime could, for example,

be Spark[1] or Flink[2].

3.1.3 Providing an Example Project

Another goal is to provide an example project that uses the library. The purpose of the example project is to evaluate the practicability of the library and provide a usage example.

3.2 Functional Requirements

A functional requirement is a requirement that specifies a function of the system by describing the input and the expected output(Roman, 1985).

3.2.1 Cloud-Native Application Design

A cloud-native approach simplifies the deployment of applications. A core feature of cloud-native applications is the usage of containers. A container is an image of an operating system with an installed and pre-configured application. At runtime, it shares the Kernel of the host system. It simplifies the deployment and update of an application (Pamidi & Vasudeva, 2015). An ETL pipeline may include many different services with complex configurations, so a container-based approach allows a steep learning curve.

3.2.2 Providing a full Data Flow

The goal of this thesis is to be able to convert a full "Data-Flow" into a Kafka environment. The Data Flow should contain an **E**xtract of data, a **T**ransformation, and the **L**oading of the data.

[1] https://spark.apache.org/
[2] https://flink.apache.org/

4 Solution Design

This chapter gives a general overview of the software design behind the kernel-to-kafka library. A more detailed description of components follows in chapter 5. In the subsequent multiple terminologies are used that may not be clear and have multiple meanings. To clarify, the following provides an explanation of those words with examples:

- **Component**: An application, library, or specification.
- **Ecosystem**: Components provided by companies or persons related to a specific component as Kafka Connect for Kafka.
- **Runtime**: Components of a specific ecosystem like Kafka configured to fulfill an ETL task.
- **Platform**: The used platform to run the components, in the context of this thesis an orchestrator like Docker Compose.

An important design decision is whether to use an online or offline configuration approach of the runtime. An online approach would be starting a defined set of Kafka components and configure them while they are running. This may happen by changing configuration files or triggering APIs that support online configuration. The problem of online configuration is the need for an existing Kafka Connect server that has all needed connectors installed, and a Kafka Streams application is needed that supports online configuration. Installing a new Kafka Connect connector requires a restart of the Kafka Connect server. In contrast, an offline configuration approach requires restarting the runtime for changes. The need to restart the complete runtime for changes can be remedied by running Kafka Connect components, and Kafka Streams components in separate containers that are restartable individually. The platform typically supports a partial update of the runtime, which results in similar behavior as an online configuration approach. For the upper reasons an offline configuration approach was chosen for the kernel-to-kafka library, but it is possible to add an online configuration functionality in the future.

4.1 Data Structures

The core of the kernel-to-kafka engine are three data structures that stepwise reduce the descriptive capabilities and generality to get a runtime- and platform-specific configuration of an ETL process. The design model is orientated at the design of a software compiler, as mentioned in the fundamentals chapter.

4.1.1 Kernel

The input data structure for the conversion is the Kernel. In the terminology of compilers, the Kernel represents the *source language*. A difference is that there is no need to tokenize and parse the Kernel as it is already present in a machine-readable, structured format. The application provides classes that allow navigation through the Kernel and allows querying implicit properties like inheritance and type relations.

4.1.2 Intermediate Representation Description

The Kernel is converted into the Intermediate Representation Description (IRD). The IRD is designed as a generic abstraction of typical components involved in an ETL process. In the terminology of compilers, this is the equivalent to the *intermediate language*. It removes abstractions from the Kernel such as packages and inheritances. The IRD allows runtime independent optimization and modification of the ETL process. The data structure itself is designed as a cycle-free tree that allows forward and backward navigation through the ETL process.

4.1.3 Concrete Representation Description

The IRD is converted into a runtime dependent data structure, the so-called Concrete Representation Description (CRD). The CRD represents a concrete configuration of a Kafka environment. In the terminology of compilers, this would be the *assembly language*. It uses a defined set of Kafka components to describe the ETL process. The data structure itself is designed as a list of components with no links and navigation properties. In the case of Kafka, the data structure defines configuration files of components.

4.1.4 Platform

Given the CRD as input, the last step is the conversion to a specific platform. In the terminology of compilers, this refers to the *machine language*. An example of a target platform is Docker Compose. The output is platform-dependent, for example, a directory on the filesystem containing a Docker Compose file and additional configuration files.

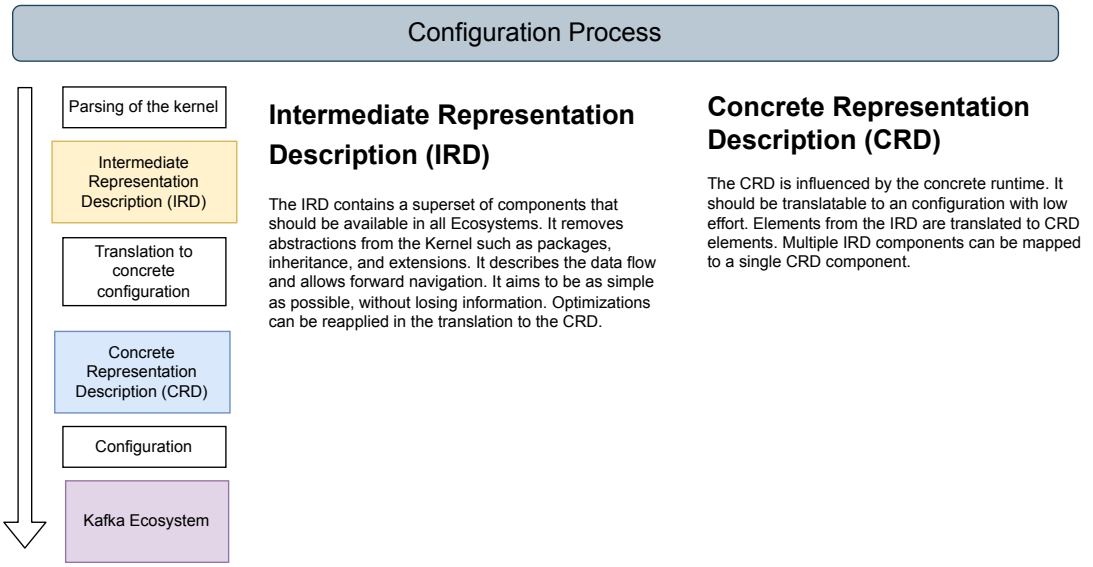


Figure 4.1: Schematic of the steps of a configuration/conversion process

4.2 Conversion Pipeline

The complexity of the different conversion steps is hidden behind a conversion pipeline. The conversion pipeline contains a list of steps executed one by one and result in the configured output. The pipeline handles the conversion between the different data-structures and contains the optimization steps.

The conversion starts with the Kernel as input, stored in its output format as a JSON file. Using the Kernel, a Kernel Reader is constructed which allows navigation through the data structure. The Reader is passed to a converter that scans the Kernel for elements that support conversion into the IRD and starts the conversion. The output of this phase is the IRD describing the ETL process.

On this representation, transformations and optimizations are applied. The next step is the conversion of the IRD to the CRD. The CRD is a list of Kafka components and their configuration. It may already contain configuration files and other additional resources.

The last step is mapping to a concrete environment. The only supported concrete environment at the moment is Docker Compose. A VFS is provided for the configuration, allowing the in-memory creation of the environment configuration, which can be stored on the target system, like the local filesystem, a remote server, or an archive.

4.3 Runtime Architecture

Every data source, transformation, and data sink is mapped to a distinct service at runtime. This allows individual scaling of services as needed by the use case. The services are packed as Docker Images for the Docker Compose platform. The images are based on existing components of the Kafka ecosystem and are extended to fulfill specific tasks specified originally by the Kernel.

5 Design and Implementation

The following chapter describes the design and implementation of the data structures presented in the previous chapter and explains the conversion process between those data structures. It also presents the design and implementation of additional Kafka Components that where needed to run ETL processes.

5.1 Technology Stack

The core language of the project is Typescript[1]. TypeScript was chosen as the primary language for the project for multiple reasons. It provides great flexibility in its execution range. Typescript can be transpiled into Javascript which can be executed by a Web Browser or server-side with NodeJS. It also provides compatibility with existing projects for the Kernel.

For Kafka Components like Connectors, Java 8 was chosen as language, and for the development of Kafka Stream Components Java 11. The reason for this is that the Kafka Connect and Kafka Stream API is provided for the Java Runtime Environment (JRE). Maven manages the build process of those components. All Kafka Components are published as container images.

Licenses of third-party dependencies used in the project are compatible with the Apache 2 license.

The library and all components related to it are provided at GitHub in the form of a monorepo, which is managed by Nx[2]. The build process of the Java projects and container images is integrated with Nx.

5.2 Kernel

As already mentioned in section 4.2 the Kernel is the input for the first phase. The library allows specifying the Kernel as a path to a file or an object containing

[1] https://www.typescriptlang.org/
[2] https://nx.dev/

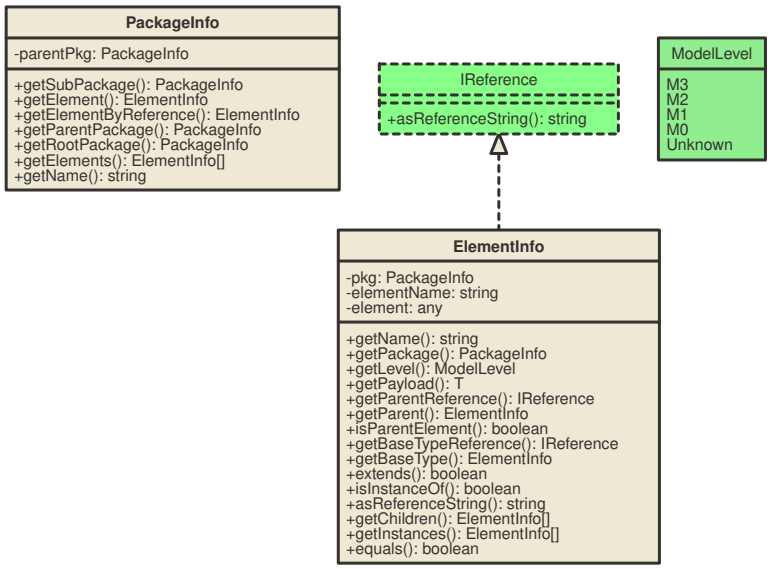


Figure 5.1: UML Diagram of the Kernel Reader

it in its export format. If the Kernel is specified as a path, the file is opened and parsed into an object containing the Kernel.

5.2.1 Kernel Reader Components

The object containing the Kernel now allows the creation of a Kernel Reader object, which is the first component of the library. Its purpose is reading and navigating through the Kernel. The core of it consists of two classes *ElementInfo* and *PackageInfo*, fig. 5.1 contains a UML description of this component. A *PackageInfo* object allows to query *ElementInfo* objects from the Kernel by providing the name and path of the elements; thus, it is possible to access known kernel elements. The Kernel Reader was created because at the time this thesis started no existing library supported reading the Kernel.

ElementInfo allows querying of related elements based on the inheritance and extension trees. The type of the element is exposed via *getBaseType()* and the object the element extends from via *getParent()*. Navigation in the other direction is also supported by the methods *getChildren()* and *getInstances()*. The implementation of those methods simply checks every element for its type or parent. A more sophisticated implementation was not chosen as a standalone library for reading the Kernel will be available in the future.

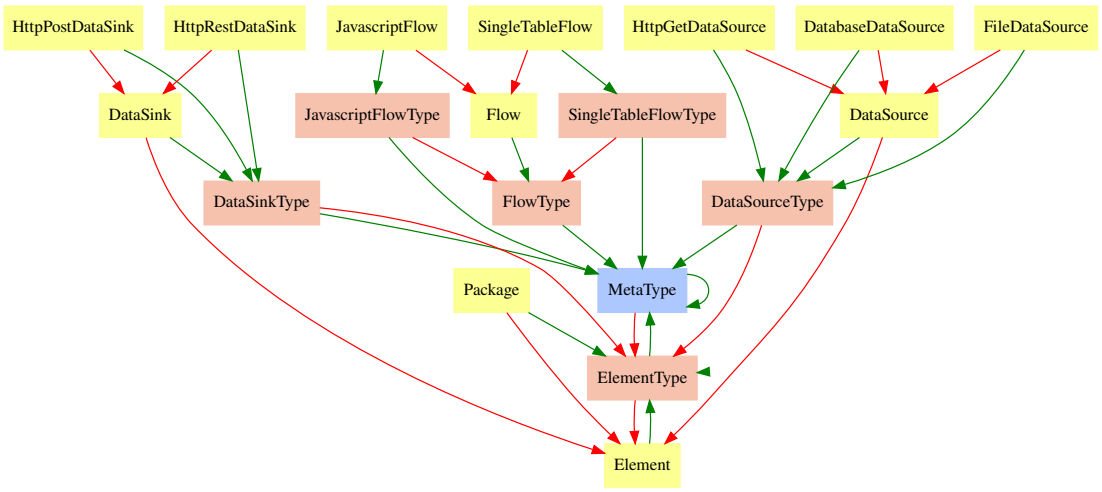


Figure 5.2: Overview of the Kernel elements introduced by the ETL extensions package, colored by their level: blue is meta-language, red is language, yellow is model, and green refers to instance level.

5.2.2 Visualize the Kernel

As shown in section 2.5 the Kernel is complex. The next section presents a newly designed kernel package for data flow. For the design of this kernel extension, a visualization was created which helps understanding the objects and structure. The kernel visualization was created with the help of the graphviz[3] library. The library provides a description language, called dot, which allows to describe directed graphs and render them as an image.

The algorithm for visualization adds every kernel element as a node, colored by the model level it is placed on. The level an element is placed on is calculated by evaluating the element's distance to the *MetaType* in the type tree. Also, edges are added based on the type and extend relations colored by the source relation it results from. A result of the visualization can be seen later in fig. 5.2. The visualization component is also included in the example web interface introduced in chapter 6.

5.2.3 Data Flow Extensions

When this thesis was created, there was no stable and feature full package for the Kernel that allows the modeling of ETL processes. To solve this problem, a Kernel package was designed to provide basic building blocks for a full ETL description, that gets converted into Kafka. The ETL extension of the Kernel contains nodes of three types: sources that represent different ways to extract

[3]https://graphviz.org/

data, flows that describe the transformation of data, and sinks that load the data into the target datastore.

During the development of the Kernel to Kafka library, the JValue project group published a new library in the modeling ecosystem that allows to define data schemas and exports them as JSON-Schema. This library was integrated into the ETL extension, which allows data sources and flows to store information about the output schema.

The design of the data flow package is not complete, it is the start of a structure that needs to be expanded and implemented further.

The following components are available in the data flow extension package:

Language Level Components

- **FlowType**: Currently has no behavior, should be used to restrict the supported sinks and sources.
- **JavascriptFlowType**: Stores the Javascript function, the input data schema, and the output data schema
- **DataSinkType**: Currently has no behavior, should be used to restrict the supported sources.
- **SingleTableFlowType**: Stores the information relevant for parsing a file containing a table
- **DataSourceType**: Currently has no behavior, should be used to restrict the supported sinks.

Model Components

- **Flow**: Stores sources, sinks, and output data schema
- **DataSink**: Stores the sources
- **DataSource**: Stores the sinks and output data schema
- **HttpRestDataSink**: Stores the path, host, and port of the provided endpoint
- **HttpPostDataSink**: Stores the endpoint the data is posted to
- **JavascriptFlow**: Currently has no behavior
- **SingleTableFlow**: Currently has no behavior
- **HttpGetDataSource**: Stores the endpoint the data is gathered from

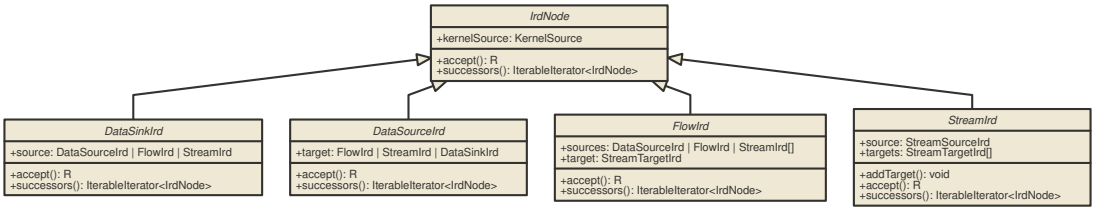


Figure 5.3: Overview of the base classes the IRD is built on.

- **DatabaseDataSource**: Stores the connection information for the database
- **FileDataSource**: Stores the path the data is gathered from

The full package in its output format can be found in appendix section A.

5.3 Intermediate Representation Description

This section presents the design and implementation of the next data structure in the conversion process. The IRD is a description model free of packages, inheritance, and extensions, describing a semantically equivalent ETL process. For the Kafka runtime, relevant elements are located in the instance level in the Kernel (the green nodes in fig. 5.2). The elements located in this level are not enough to build the Kafka runtime. There is also relevant information stored at the model level. In the case of the JavaScript flow, the *IdentityJsFlow* elements contain the function text of the JavaScript function used for the transformation. The Kernel requires complex navigation to get all information needed to implement an ETL-element. The IRD reduces this complexity.

5.3.1 Components

The structure of the IRD is orientated at the different ETL steps. A base type is provided for every step, which provides the linking logic for the data flow.

- **DataSinkIrd**: A node that represents a load operation of data. Supports a single source.
- **DataSourceIrd**: A node that represents a extract operation of data. Supports a single target.
- **FlowIrd**: A node that represents a transformation of data. Supports multiple sources and a single target.
- **StreamIrd**: A node that allows multicasting of data. Supports a single source and multiple targets.

- **DataSchemaIrd**: A node that stores a data schema.

Based on those based types the following components are available:

- **FileDataSource**: A node that represents a data extraction out of a file. The file is read line by line, the output format is of type string.
- **HttpDataSink**: A node that represents data loading by performing a HTTP request. The data format is defined by its input.
- **HttpDataSource**: A node that represents a data extraction by performing a HTTP request. The response is read line by line, the output format is of type string.
- **JavascriptFlow**: A node that represents a data transformation by executing a provided javascript function. The input format is defined by its input. The output format is specified explicitly.
- **Multicast**: A node that represents a multicast operation on data. It shares data with multiple outputs.
- **RestDataSink**: A node that represents data loading by providing an REST endpoint. The data format is defined by its input.
- **SingleTableFileFlow**: A node that represents a data transformation that parses a string into an array of values.
- **JsonDataSchema**: A node that represents a JSON schema definition,
- **StringDataSchema**: A node that represents a string data schema.

The goal of the IRD is to provide an easy way for navigating through the data flow. This is achieved by a function of every IRD node that allows iteration over its successors. A successor is a node where data flows to.

The root object of the IRD contains a list of data sources. All relevant nodes can be reached by starting the navigation from the data sources.

5.3.2 Conversion from the Kernel

For the Kernel conversion into the IRD an extensible approach was chosen. A converter has two methods: *canHandle* and *convert*.

```
1 export class JavascriptFlowConverter extends
    KernelIrdNodeConverter {
2   canHandle(element: ElementInfo): boolean {
3     ...
4   }
5   async convert(element: ElementInfo, manager: ConvertManager):
    Promise<Result<IrdNode>> {
6     ...
```

```
  }
}
```

For the conversion, every kernel element is iterated, and every converter is asked if handing of this node is possible. In most cases *canHandle* checks if the node is an instance of a model level Kernel element it supports.

```
canHandle(element: ElementInfo): boolean {
  return element.isInstanceOf(KernelReferences.JavascriptFlow);
}
```

The *convert* function gets the element it should convert to an IRD node and a context object as input. The context allows the converter to wait for the conversion result of other Kernel elements.

The first step in the conversion is extracting the payload attached to the Kernel element. The payload is all of the kernel element's data, like a name or connected sinks. The payload type information is provided by the Data Flow Extensions package, as the payload equals the export type definitions.

```
/* Stores the source and sink of the flow */
const jsFlowPayload = element.getPayload<JavascriptFlowJsonExport
    >();

/* Stores function and schema definition */
const jsFlowTypeElement = element.getBaseType();
const jsFlowTypePayload = jsFlowTypeElement.getPayload<
    JavascriptFlowTypeJsonExport>();
```

Component-specific information is directly extracted out of the payloads and transferred to the IRD node. This includes the data schema if the Kernel node provides a data schema.

As the IRD stores forward data flow information, a reference to all succeeding nodes needs to be acquired. For this, the *ConvertManager* can query the result of the conversion of those elements.

```
const targets: StreamTargetIrd[] = [];
for (const sink of jsFlowPayload.sinks) {
  const ele = element.getPackage().getRootPackage().
    getElementByReference(new SimpleReference(sink));
  const nodeResult = await manager.waitForNode(element, ele);
  // error handling
  targets.push(nodeResult.data as StreamTargetIrd);
}
```

If the conversion result of the requested element is already available, it is returned as a resolved promise. If an element is not available yet, a promise for the conversion result is returned, and the current element is entered in a waiter graph

as waiting for the other kernel element. If the waiter graph contains a cycle, an exception is thrown as the conversion isn't going to finish.

```
waitForNode(requester: ElementInfo, target: ElementInfo): Promise
    <Result<IrdNode>> {
  ...
  this.waiterGraph.addEdge(requester, target);
  const cycle = this.waiterGraph.isCyclic(true);
  if (cycle) {
    // throw error
  }
  return targetFuture.toPromise();
}
```

The additional check of the waiter graph simplifies the development of new converters as in case of a deadlock concrete feedback is given where the cycle is located.

The order of the conversion is implicitly managed by the JavaScript runtime. For this, an completable future type, as available in Java, is implemented, which allows resolving a promise as soon as a value is available. If the conversion depends on the result of another node, the JavaScript runtime will continue the conversion of other nodes till the result is available. This method allows the conversion to make efficient use of asynchronous operations. The usage of those operations may be necessary if the kernel is read asynchronous.

During the conversion, all generated nodes that are a data source are collected, as they provide the entry point into the IRD tree structure. Converted nodes that are not connected to a data source are collected by the garbage collector and are not accessible after the conversion into the IRD.

For every created IRD a reference to the source of the Kernel is stored. This allows giving the user precise error locations in case of failures in the optimization or later conversion processes.

```
const jsFlowIrd = new JavascriptFlowIrd(...,
    SingleNodeKernelSource.from(element));
```

5.3.3 Optimizations

For the support of optimization on the IRD several helping data classes are implemented. Depending on the optimizer's goal, the two most essential helper classes are a Visitor and a ForwardIterator. The ForwardIterator uses the successors property of the *IrdNode* class to acquire a list of successors and recursively iterates over the returned elements.

```
private *traverseNode(node: IrdNode): IterableIterator<IrdNode> {
  yield node; // return the current node
```

```
3   const successors = node.successors();
4   let successor: IteratorResult<IrdNode>;
5   while (!(successor = successors.next()).done) {
6     yield* this.traverseNode(successor.value); // depth first
      traversal
7   }
8 }
```

The provided visitor is an extension of the visitor provided by the Gang of Four(Patterns, 1995, pp. 331). It supports a prolog and epilog that is executed before every node and has virtual members that are called before a child is visited and after a child is visited. An example that uses those helper functions can be found in the *MulticastIrdOptimizer*, which ensures that all flow nodes output their results in a multicast node.

```
1 export class MulticastIrdOptimizer extends BaseIrdVisitor<void,
    void> implements IrdOptimizer {
2   protected prolog(node: IrdNode): void {
3     if (node instanceof FlowIrd) {
4       if (!(node.target instanceof MulticastIrd)) {
5         node.target = new MulticastIrd([node.target], node.
    kernelSource);
6       }
7     }
8   }
9 }
```

This optimizer uses the *prolog* called before a node is further processed and checks if the node that is currently visited is of type *FlowIrd*. For the case the target is is a multicast node nothing is done else the target is replaced by a multicast node.

5.4 Concrete Representation Description

The next data structure in the conversion process is the CRD. The CRD is specific to the target ecosystem but contains a generic base class for every node. This would be equivalent to the assembly language in the classical compiler model. All relevant information for an ecosystem component is already present, like configurations. Still, it is not directly executable until converted into a platform output (the machine language in the compiler analogy). The generated CRD depends on the IRD and an additional configuration that manages aspects that are not stored in the IRD, such as the generation of additional debugging components.

The CRD itself should not be considered as graph data structure. The creation of a node in the list may depend on another node for the creation, but this dependency should be implicit as the conversion to a platform does not ensure a

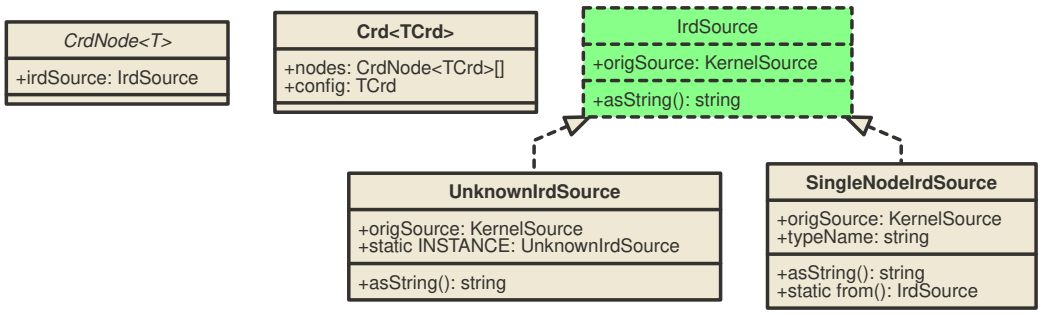


Figure 5.4: UML structure showing the base classes of the CRD

processing order of CRD nodes.

There is no one-to-one mapping of an IRD node to a CRD node. There are cases where one CRD node may handle the task of multiple IRD nodes or the other way round.

5.4.1 Generic Components

The base class defines a property for tracking the IRD origin of the node. This property is helpful for the error handling of the conversion abstraction presented later.

```
export abstract class CrdNode<T extends CrdType> {
  readonly irdSource: IrdSource;

  protected constructor(irdSource: IrdSource) {
    if (!irdSource) {
      throw new Error('irdSource must be defined');
    }
    this.irdSource = irdSource;
  }
}
```

Also, a class CRD holds the list of nodes and an additional configuration used to build the CRD.

```
export class Crd<TCrd extends CrdType> {
  constructor(
    public readonly nodes: CrdNode<TCrd>[],
    public readonly config: TCrd
  ) {}
}
```

5.4.2 Kafka Specific Components

For the Kafka-specific CRD two main base classes exist: one for Kafka Streams component and one for Kafka Connect components. The Kafka Connect base class declares the worker and connector configuration needed for the component. For this, a data structure that represents the configuration files was introduced. Kafka configuration files are key-value files.

```
export class Properties {
  store = new Map<string, string>();
  public get(key: string): string | undefined {
    return this.store.get(key);
  }
  ...
  public asString() {
    return Array.from(this.store.entries())
      .map((entry) => `${entry[0]}=${entry[1]}`)
      .join('\n');
  }
}
```

The Kafka Connect node declares one *Properties* instance for the worker configuration and one for the connector configuration and exposes common properties via getter and setters.

```
export abstract class KafkaConnectCrd extends CrdNode<
   KafkaCrdType> {
  protected workerProperties = new Properties();
  protected connectorProperties = new Properties();

  set bootstrapServers(value: string) {
    this.workerProperties.set('bootstrap.servers', value);
  }
  ...
}
```

Concrete connectors use those setters to configure the basic configuration and expose specific setters for connector-specific configurations.

```
export class CastormKafkaConnectHttpCrd extends KafkaConnectCrd {
  constructor(name: string, target: TopicCrd, ...) {
    super(irdSource);
    this.name = name;
    this.bootstrapServers = kafka.serverUrl;
    ...
  }
  set topic(topic: string) {
    this.connectorProperties.set('kafka.topic', topic);
  }
  ...
}
```

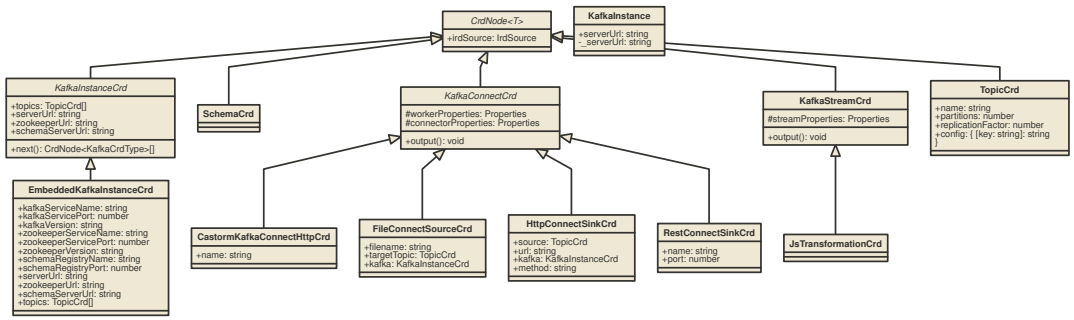


Figure 5.5: UML structure of the Kafka specific nodes available in the CRD structure

Sink Connectors and Stream components always get their data out of topics, Source Connectors and Stream components always store their data into topics. In the CRD topics are represented by a node that stores the format of the data in the topics and provides the according converter for the data format to the components. If supported CRD nodes are provided the output schema of the data to perform a validation.

```
1
2  export class TopicCrd extends CrdNode<KafkaCrdType> {
3    name: string;
4    valueType: ValueTypes;
5    ...
6    getValueConverter(): string {
7      switch (this.valueType) {
8        case 'json':
9          return 'org.apache.kafka.connect.json.JsonConverter';
10       case 'json-schema':
11         return 'io.confluent.connect.json.JsonSchemaConverter';
12       case 'string':
13         return 'org.apache.kafka.connect.storage.StringConverter'
       ;
14      }
15   }
16   ...
17 }
```

The following other nodes exists in the CRD:

- **CastormKafkaConnectHttp**: Combines the HttpDataSource and SingleTableFile into one Kafka Connector
- **EmbeddedKafkaInstance**: Represents a Kafka Instance. The creation of this node depends on the additional configuration used for CRD creation.
- **FileConnectSource**: Represents a source file that is imported line by line

from file system.

- **CamelPostgresConnectSource**: Represents a source that gathers data by executing a Structured Query Language (SQL) query.
- **HttpConnectSink**: Represents a sink where data is posted to.
- **JsTransformation**: Represents a JavaScript transformation.
- **CsvTransformation**: Represents a String to JSON array transformation.
- **RestConnectSink**: Represents a sink that provides the data as an REST endpoint.

5.4.3 Conversion from the IRD

The next phase in the conversion process is the CRD conversion. It takes as input the IRD and additional configuration hints for the target environment and produces a description of all resources in Kafka. The challenge in this phase is that not all IRD nodes can be directly mapped to a CRD node. A simple example of this problem is the import of a CSV file from a web server. The IRD contains a *HttpDataSource* and a *SingleTableFileFlow* node, in Kafka a single Kafka Connect source can do the work. The following two solutions for converting the IRD to the CRD are presented. Both of them have benefits and problems. It is unclear how the benefits and problems are weighed as it depends on the future use of this library.

Responsibility Based Conversion

The responsibility-based converter uses a similar interface used in the Kernel to IRD conversion. The converter has a list of converters that provide a method *canHandle*, *getDependencies*, and *convert*. The selection of a converter to a part of the IRD works like pattern matching. The *canHandle* checks if a node pattern it supports occurs at a specific location in the IRD. If it can convert a subtree of the IRD to the CRD it tells the conversion manager the exact part it is able to handle.

```
canHandle(irdNode: IrdNode): CanHandleResult {
  if (!(irdNode instanceof SingleTableFileFlowIrd))
    return CanHandleResult.No();
  const httpDataSourceIrd = irdNode.sources[0];
  if (!(httpDataSourceIrd instanceof HttpDataSourceIrd)) {
    return CanHandleResult.No();
  }
  return CanHandleResult.Yes([irdNode, httpDataSourceIrd]);
}
```

The example checks if the IRD contains the following pattern *HttpDataSourceIrd* $\longrightarrow$ *SingleTableFileFlowIrd*. If so, it returns that it can handle both nodes' conversion. The *convert* method will be called with the same IRD node as a parameter where the yes result was returned.

The conversion manager tries to find converters for the whole IRD tree. For this a list of all nodes in the IRD is generated using the ForwardIterator.

```
const irdNodes: IrdNode[] = Array.from(new ForwardIrdIterator(
    input));
```

If it cannot apply a converter at any subtree of IRD the conversion is aborted.

The next step in the conversion is the building of a dependency tree. For this, every converter is asked to provide its dependencies. A dependency can be an IRD node that needs to get converted first or a particular object, like the Kafka server.

```
getDependencies(
  context: CrdDependencyContext,
  node: HttpDataSourceIrd
): (IrdNode | NamedCrdNode<CrdNode<KafkaCrdType>>)[] {
  return [node.target, DefaultKafka];
}
```

The example shows the dependencies of the HttpDataSource node. It depends on the target, as it needs to know in which topic the data is output and the Kafka instance it works on. The conversion manager checks if there are any cycles in the dependency tree and, if so, cancels the conversion as it is impossible.

The IRD nodes are converted it the topological order acquired from the dependency tree.

```
const nodes = depGraph.getTopologicalSort();
```

This ensures that all dependencies are ready for every converter when the node is converted. As a result, the conversion returns a primary CRD node and an optional list of secondary nodes. The primary result is linked with the original IRD node, which allows other converters to query those CRD by the IRD node.

Visitor Based Conversion

Another strategy for converting the IRD to the CRD is using a visitor-based approach. The IRD visitor executes a method, based on the node type, for every node of the IRD in a depth-first manner. The method is responsible for passing all successor nodes to the visitation. This implies that at the conversion of an IRD node, the conversion result for the whole subtree starting at a node is available but not necessarily the conversion of the parent node. The pattern

matching approach described in the upper conversion strategy is also possible in the visitor-based strategy. For this, at the visitation of the first node in the pattern, it is checked if the pattern matches.

```
visitHttpGetDataSource(node: HttpDataSourceIrd): CrdNode<
    KafkaCrdType> {
     const conversionStep = node.target;
     ...
     if (conversionStep instanceof SingleTableFileFlowIrd) {
       const target = conversionStep.target;
       ...
       target.accept(this, undefined);
}
```

If the other nodes in the pattern are not convertible on their own, error handling can be done by simply throwing an error if the node is visited.

```
visitSingleTableFileFlow(node: SingleTableFileFlowIrd): CrdNode<
    KafkaCrdType> {
  this.errors.push(‘SingleTableFileFlowIrd should have
    HttpDataSourceIrd as source‘);
  return null;
}
```

If the pattern matches, the visitation of the node is skipped.

Comparison of the Conversion Strategies

Benefit of the responsibility based conversion presented in section 5.4.3 is the extensibility. The conversion library can be easily extended by plugins which only need to register new converters in the conversion manager. The benefit of the visitor-based conversion presented in section 5.4.3 is the simplicity, and the needed implementation effort is much lower. An example of this is the omitted topological sort of the nodes.

A problem that occurs with both conversion strategies is the generation of names. Names are needed for Kafka Components, streams, and services created by the orchestrator. A simple naming strategy is the generation of random names. This approach has the problem that the generated output is not deterministic, which prevents partial updates executed by the orchestrator and results in a regeneration of streams every time the runtime is updated. This problem is solved by using a pseudo random generator that uses the nodes source as input.

```
nextPseudoRandomName(seed: T, prefix = '', postfix = ''): string
    {
  let prng = this.randomProvider.get(seed);
  ...
  let name: string;
  do {
```

```
6       name = prefix + prng().toString(36).substring(2, 7) + postfix
        ;
7     } while (this.generatedNames.has(name));
8     this.generatedNames.add(name);
9     return name;
10  }
```

5.5 Platform

The last phase in the conversion of the Kernel to Kafka is the configuration of a concrete platform. The input for this phase is a list of Kafka resources, the so-called CRD. One way of creating Kafka components would be to start the services directly. Problems that may arise are replication of services, updating services, including data persistence, and supporting different operating systems and their configurations. Luckily, orchestrators have already solved this problem, as presented in the fundamentals section. The output of this phase is the input for an orchestrator. The library includes an implementation for the Docker Compose orchestrator.

5.5.1 Virtual File System

An orchestrator usually takes files located in the file system as input. The usage of the file system in a library comes with three major problems:

- The machine the library runs on may not be the same the orchestrator runs on.
- In specific environments, like a web browser, no file system is available.
- Additional error handling is needed for file system interaction.

An abstraction of the file system was designed to solve those problems, the VFS. The VFS is a tree structure consisting of two node types: folders and files. Folder nodes have a name and a list of child nodes, whereas file nodes are leaves and only have a name and a buffer with the file's content. This allows the creation of all files needed by the orchestrator without depending on outputting them directly. Figure 5.6 provides a visualization of the VFS's class structure.

5.5.2 Docker Compose Platform

The output for the Docker Compose platform is a directory VFS node containing the *docker-compose.yaml* file, which is the main input for the orchestrator, and additional nodes containing configuration files. For writing the VFS into the native file system, the user can use the existing *FsVfsWrite*, visible in fig. 5.6 or provide an own implementation of an *VfsWriter*.

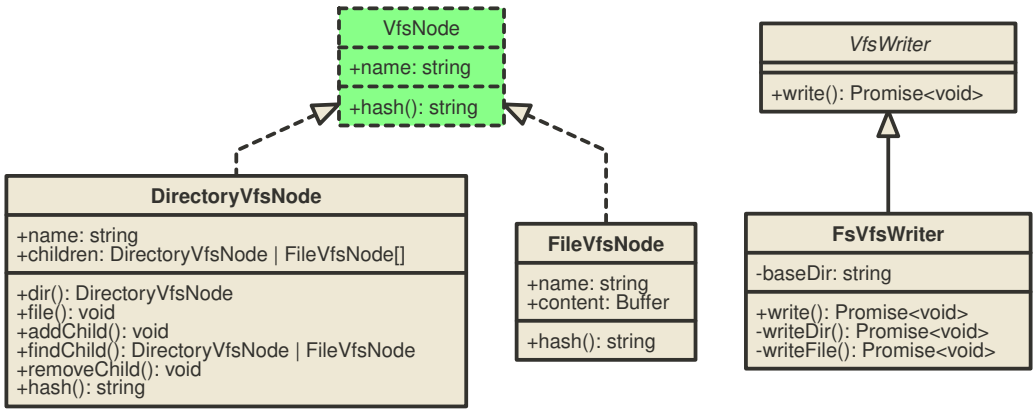


Figure 5.6: UML graph showing classes of the VFS.

5.5.3 Conversion from the CRD

The conversion from the CRD to the Docker Compose platform is a straightforward process. For every type of node that occurs in the CRD, a configurator class exists, which adds services, networks, and files to the Docker Compose Platform. The configurator class has access to the VFS, which is used to write configuration files and an object with the Docker Compose specification (the *docker-compose.yaml* file). As the services may have volumes containing the configuration files, a hash of those files is appended to the service's labels. This causes the orchestrator to recreate the service every time the configuration changes. Last step of the conversion is the output of the Docker Compose specification in the VFS.

The implementation of a new ecosystem requires writing new configurator classes. As the conversion depends on the CRD, the conversion needs to be implemented for every platform-runtime combination.

It would also be possible to implement an conversion process that directly configures the host system's services. This would represent an online configuration.

5.5.4 Kafka Streams - CSV Transformation

For the *SingleTableFileFlow* IRD no existing Kafka component fitted. To support this node a custom implementation of an application using the Kafka Streams API is provided. The application takes a Kafka Topic as input and output and reads the messages in string format. The string message is processed using the opencsv library[4] and returned as an JSON array.

[4]http://opencsv.sourceforge.net/

5.5.5 Kafka Streams - JavaScript Transformation

For the *JavascriptFlow* IRD aswell no existing Kafka comonent fitted and a Kafka Streams application was implemented.

A JavaScript function is triggered in the map step in the topology of the Streams application.

```
final StreamsBuilder builder = new StreamsBuilder();
builder
  .stream(this.config.getInputTopic(), Consumed.with(Serdes.
   String(), jsonSchemaSerde))
  .map((key, value) -> {
    /* call JavaScript function */
  })
  .to(this.config.getOutputTopic(), Produced.with(Serdes.String()
   , jsonSchemaSerde));
this.topology = builder.build();
```

The provided JavaScript function is executed by the Nashorn[5] engine. The function gets the key and value of the message as input and must return a new key and value for the message.

```
function transform(key, value) {
  return [
    key,
    {
      continent: value[1],
      total_cases: Number(value[4]) || -1,
      total_deaths: Number(value[7]) || -1
    }
  ];
}
```

5.5.6 Kafka Connect - Http Data Source

The Kafka Connect HTTP Connector used for the library is based on an open-source connector published by castorm[6] on GitHub.

The connector is internally uses a two layer architecture. The first layer parses the HTTP response into a object consisting of a key and a value. The next layer transforms the key value object into the message for Kafka.

By default the library supports reading JSON data sources. As the Kernel Data Flow package also supports CSV, the library was expanded to support reading CSV sources. This extension happens on the first layer of the connector.

[5] https://openjdk.java.net/projects/nashorn/
[6] https://github.com/castorm/kafka-connect-http

The first layer outputs a key of type string and a JSON string value. As the Kernel allows to pass a schema for output object, the second layer was extended to support the validation of this schema.

5.5.7 Kafka Connect - Rest Sink

The Kafka Rest Sink is a connector that collects the data and provides a REST interface to query the data. The goal of this sink is to provide a simple way to see result of the ETL pipeline. The Rest Sink implements the Kafka Connector API and Java Servlet API together with a Jetty web server. If a new message is published it is stored in an in memory data structure. The rest server serves messages based on the messages key. If a new message arrives the old message with this key is superseded.

6 Evaluation

The evaluation reflects upon the requirements defined earlier. The first requirement was the usage of existing Kafka Components. This was partially possible. It was possible to use existing HTTP and file data sources components. The HTTP needed to be extended to support non JSON values. Implementation of additional data sources is also possible. There exists an extensive variety of connectors for Kafka, for example, the Apache Camel[1] connectors. Those connectors often have limitations. For example, the File Transfer Protocol (FTP) connector of the Camel project looks for new files and passes the content of those into messages but does not support watching a single file on an FTP server. Fetching data from structured data sources like databases is usually no problem. A more complex problem is the implementation of the transformation with existing components. No flexible open-source solution was found, and the transformations were self-implemented.

The following requirement was the extensibility of the library to other runtimes. This is possible through the layered design. A new runtime can be added by implementing a new CRD and platform. Optimizations and transformations of the ETL process can be shared between different CRDs by applying them to the IRD. Kafka, for example, does not support a direct connection between two components; there needs to be a stream in-between always. An IRD optimizer solves this. If other runtimes have the same restriction, the optimizer can be reused. Also, the emulation of not supported transformation could be done on the IRD layer. If a target platform supports no native CSV transformation but a JavaScript transformation, the CSV step could be replaced by a slower but working JavaScript step.

Another requirement was to provide a sample project that uses the library. A project was implemented with an Angular-based frontend and a NestJS based backend. The backend application provides API endpoints to manage projects consisting of a name, Kernel, and CRD configuration. The application can execute the projects internally. It performs a conversion into the platform output and then calls Docker Compose to start the project.

[1] https://camel.apache.org/camel-kafka-connector/1.0.x/reference/index.html

```
// simplified start process of a project
ird = await project.createIrd();
crd = project.createCrd();
platform = await converter.build(crd);
const writer = new FsVfsWriter(projectDir);
await ecosystem.write(writer);
await this.projectService.startProject(project);
```

Also, endpoints are available that return visualizations of the Kernel and CRD. The frontend provides a convenient way of using those API endpoints. This shows that the library can be used in an application. An screenshot of the example application is available in appendix section B.

The cloud-native design requirement was achieved by creating a cloud-native output. The library itself is independent of resources as a file system, this allows the library to be used in many kinds of applications, for example, web browsers, serverless computing, or classical applications. For that reason, the library is also runnable in cloud environments.

The last requirement was to provide a full data flow. For this, a COVID 19 data set in CSV format is imported and transformed into JSON objects, which are loaded into a web server that provides those via REST endpoints. An examples library provides the Kernel describing this process as part of the monorepo. The generated Docker Compose file can be found in appendix section C.

7 Conclusions

The conversion of a Kernel into a Kafka runtime is possible by mixing existing and newly created Kafka components. The practicability of the implemented conversion process with existing components out of the Kafka ecosystem depends on Kernel's final design of the data flow package.

If the Kernel allows high configuration flexibility, it may be hard to use existing components. The thesis was based on a small data flow package without high configuration flexibility and already needed to extend existing libraries to support the small set of operations. This was visible in the extract and transform steps. No existing connector could read non JSON data via an HTTP request. Also, there is no flexible solution for building the transform steps. The CSV transformation was as well implemented for the library. kSQL[1] may provide a solution for the transformation problem but is not released under an Open Source license.

A reason for this is that Kafka is designed to be integrated into existing applications and micro-services. It is easier and more flexible to program a specialized Kafka adapter in those applications than with a configuration-based method.

Another approach for a Kafka runtime would be using specialized Components that natively understand the Kernel. The benefits of this approach are complete control over the components and a Kernel-orientated design that reduces the complexity of the configuration variants and thus possible errors and unsupported descriptions. The design of such runtime could provide a generic interface for producing and consuming messages. This would allow the support of additional runtimes by implementing the generic interfaces.

[1] https://github.com/confluentinc/ksql

Appendices

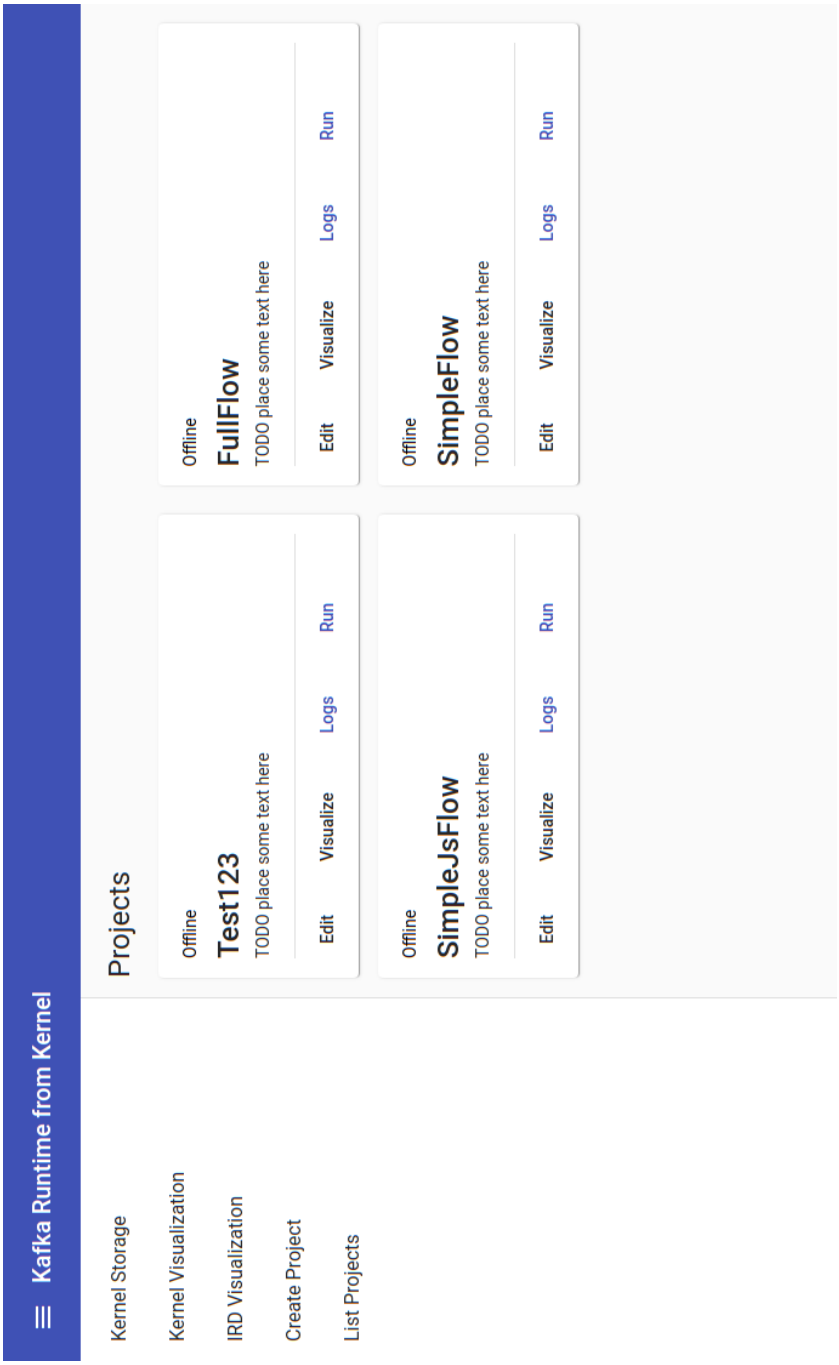
A Kernel Output of Data Flow Extension Package


```
{
    "Flow": {
      "elements": {
        "MetaValue": {
          "schema": {
            "type": "object",
            "allowAdditionalProperties": [
              {
                "type": "union",
                "elements": [
                  {
                    "type": "string",
                    "restrictions": []
                  },
                  {
                    "type": "number",
                    "restrictions": []
                  }
                ]
              }
            ],
            "properties": {}
          },
          "type": "#/System/ValueType",
          "extends": "#/System/Value"
        },
        "AnyMetaValue": {
          "schema": {
            "type": "object",
            "allowAdditionalProperties": [
              {
                "type": "union",
                "elements": [
                  {
                    "type": "string",
                    "restrictions": []
                  },
                  {
                    "type": "number",
                    "restrictions": []
                  }
                ]
              }
            ],
            "properties": {}
          },
```

```
      "type": "#/System/ValueType",
      "extends": "#/Flow/MetaValue"
    },
    "FlowType": {
      "name": "FlowType",
      "type": "#/System/MetaType"
    },
    "Flow": {
      "name": "Flow",
      "type": "#/Flow/FlowType"
    },
    "SingleTableFlowType": {
      "extends": "#/Flow/FlowType",
      "name": "SingleTableFlowType",
      "type": "#/System/MetaType"
    },
    "SingleTableFlow": {
      "columnTypes": [],
      "extends": "#/Flow/Flow",
      "name": "SingleTableFlow",
      "type": "#/Flow/SingleTableFlowType"
    },
    "JavascriptFlowType": {
      "extends": "#/Flow/FlowType",
      "name": "JavascriptFlowType",
      "type": "#/System/MetaType"
    },
    "JavascriptFlow": {
      "extends": "#/Flow/Flow",
      "name": "JavascriptFlow",
      "type": "#/Flow/JavascriptFlowType"
    },
    "DataSinkType": {
      "extends": "#/System/ElementType",
      "name": "DataSinkType",
      "type": "#/System/MetaType"
    },
    "DataSink": {
      "extends": "#/System/Element",
      "name": "DataSink",
      "type": "#/Flow/DataSinkType"
    },
    "HttpPostDataSink": {
      "extends": "#/Flow/DataSink",
      "name": "HttpPostDataSink",
      "type": "#/Flow/DataSinkType"
    },
    "HttpRestDataSink": {
      "extends": "#/Flow/DataSink",
      "name": "HttpRestDataSink",
      "type": "#/Flow/DataSinkType"
```

```
        },
        "DataSourceType": {
          "extends": "#/System/ElementType",
          "name": "DataSourceType",
          "type": "#/System/MetaType"
        },
        "DataSource": {
          "extends": "#/System/Element",
          "name": "DataSource",
          "type": "#/Flow/DataSourceType"
        },
        "HttpGetDataSource": {
          "extends": "#/Flow/DataSource",
          "name": "HttpGetDataSource",
          "type": "#/Flow/DataSourceType"
        },
        "FileDataSource": {
          "extends": "#/Flow/DataSource",
          "name": "FileDataSource",
          "type": "#/Flow/DataSourceType"
        },
        "DatabaseDataSource": {
          "extends": "#/Flow/DataSource",
          "name": "DatabaseDataSource",
          "type": "#/Flow/DataSourceType"
        }
      },
      "packages": {},
      "name": "Flow",
      "type": "#/System/Package"
    }
  },
  "name": "#",
  "type": "#/System/Package"
}
```

B Screenshot of the Example Web Application



C Docker Compose File of the COVID-19 Example

```
version: "3.4"
services:
  zookeeper:
    image: wurstmeister/zookeeper:latest
    labels:
      ods-ems.source-node: EmbeddedKafkaInstanceCrd
  kafka:
    image: wurstmeister/kafka:2.13-2.8.1
    depends_on:
      - zookeeper
    environment:
      KAFKA_CREATE_TOPICS: d5yuy:1:1,7v89h:1:1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENERS: INSIDE://:9092
      KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
    labels:
      ods-ems.source-node: EmbeddedKafkaInstanceCrd
    ports:
      - target: 9092
        published: 9092
  kafka-ui:
    image: provectuslabs/kafka-ui
    environment:
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
      KAFKA_CLUSTERS_0_ZOOKEEPER: zookeeper:2181
      KAFKA_CLUSTERS_0_SCHEMAREGISTRY: http://karapace-registry
   :8081
    ports:
      - target: 8080
        published: 8080
  karapace-registry:
    image: ghcr.io/aiven/karapace
    entrypoint:
      - /bin/bash
      - /opt/karapace/start.sh
      - registry
    environment:
      KARAPACE_ADVERTISED_HOSTNAME: karapace-registry
      KARAPACE_BOOTSTRAP_URI: kafka:9092
      KARAPACE_PORT: 8081
      KARAPACE_HOST: 0.0.0.0
      KARAPACE_CLIENT_ID: karapace
      KARAPACE_GROUP_ID: karapace-registry
      KARAPACE_MASTER_ELIGIBILITY: "true"
      KARAPACE_TOPIC_NAME: _schemas
      KARAPACE_LOG_LEVEL: WARNING
```

```
      KARAPACE_COMPATIBILITY: FULL
    ports:
      - target: 8081
        published: 8081
  http-connect-source-2i9dm:
    image: jvalue-connect
    depends_on:
      - kafka
    volumes:
      - type: bind
        source: ./http-connect-source-2i9dm/worker.properties
        target: /worker.properties
      - type: bind
        source: ./http-connect-source-2i9dm/connect.properties
        target: /connect.properties
    labels:
      ods-ems.source-node: CastormKafkaConnectHttpCrd
      ods-ems.config-hash: 2491d1630bf1a9449ea297a26c2d67a...
  js-trafo-m49s1:
    image: kafka-js-trafo
    depends_on:
      - kafka
    environment:
      START_CLASS: de.jvalue.kafka.JsTransform
    volumes:
      - type: bind
        source: ./js-trafo-m49s1/stream.properties
        target: /stream.properties
    labels:
      ods-ems.source-node: JsTransformationCrd
      ods-ems.config-hash: 0dc45992c59e265053e9de40a971f2a59...
  rest-connect-sink-2oibb:
    image: kafka-rest-sink
    depends_on:
      - kafka
    ports:
      - 1234:1234
    volumes:
      - type: bind
        source: ./rest-connect-sink-2oibb/worker.properties
        target: /worker.properties
      - type: bind
        source: ./rest-connect-sink-2oibb/connect.properties
        target: /connect.properties
    labels:
      ods-ems.source-node: RestConnectSinkCrd
      ods-ems.config-hash: 4c707a9b0f6627d76fccd96dcac98a57c...
```

References

Aho, A., Lam, M., Sethi, R. & Ullman, J. (2007). *Compilers: Principles, techniques, & tools*. Pearson/Addison Wesley. https://books.google.de/books?id=VkLKhpAXzrsC

Apache Kafka powerby [Accessed: 2022-03-27]. (n.d.). https://kafka.apache.org/powered-by

Kimball, R. & Caserta, J. (2011). *The data warehouse etl toolkit: Practical techniques for extracting, cleaning, conforming, and delivering data*. Wiley. https://books.google.de/books?id=TCLfzU2ilVkC

Narkhede, N., Shapira, G. & Palino, T. (2017). *Kafka: The definitive guide : Real-time data and stream processing at scale*. O'Reilly Media. https://books.google.de/books?id=qIjQjgEACAAJ

Pamidi, M. & Vasudeva, A. (2015). Impact of containers on data center virtualization. *Website: http://www.itnewswire.us/Containers.pdf, diakses tanggal*, *24*.

Patterns, D. (1995). Elements of reusable object-oriented software.

Roman. (1985). A taxonomy of current issues in requirements engineering. *Computer*, *18*(4), 14–23. https://doi.org/10.1109/MC.1985.1662861