



Big Data for Mobility Tracking Knowledge Extraction in Urban Areas

D2.3 Development of Toolboxes Integration Connectors

Document Summary Information

Grant Agreement No	780754	Acronym	TRACK & KNOW
Full Title	Big Data for Mobility Tracking Knowledge Extraction in Urban Areas		
Start Date	01/01/2018	Duration	36 months
Project URL	https://trackandknow.eu		
Deliverable	D2.3 Development of Toolboxes Integration Connectors		
Work Package	WP2 Data Processing Architecture & Infrastructure (BDMI Toolbox)		
Contractual due date	20/06/19	Actual submission date	20/06/19
Nature	Other	Dissemination Level	PU
Lead Beneficiary	05 - INTRASOFT		
Responsible Author	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)		
Contributions from	Toni Staykova (CEL), Leonardo Longhi (SIS), Fabio Manichetti (CNR), Mirco Nanni (CNR), Gennady Andrienko (Fraunhofer), Ian Smith (PAP), Akrivi Vlachou (UPRC), Christos Doulkeridis (UPRC), Yannis Theodoridis (UPRC), Athanasios Koumparos (VFI), Anagnostis Delkos (VFI), Panos Livanos (VFI)		



This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 780754.

Revision history (including peer reviewing & quality control)

Version	Issue Date	% Complete	Changes	Contributor(s)
V0.1	15/04/19	5%	Initial Deliverable Structure	Ioannis Daskalopoulos (INTRA)
V0.2	13/05/19	85%	Initial Internal Review version	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)
V0.3	17/05/19	95%	Internal Review version	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)
V0.4	05/06/19	99%	Peer Review Contributions	Ian Smith (PAP), Kieran Lee (PAP), Nikos Katzouris (NCSR)
V0.5	10/06/19	100%	QA process	Marios Logothetis (INTRA)
V1.0	19/06/19	100%	Final version	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)
V1.1	27/08/19	100%	Final version with Annex	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)

Disclaimer

The content of the publication herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the TRACK&KNOW consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the TRACK&KNOW Consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the TRACK&KNOW Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© TRACK&KNOW Consortium, 2018-2020. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Executive Summary

This document extensively addresses Track&Know Development of Toolboxes Integration Connectors based on requirements, facilitating the seamless integration of internal and external data sources. The scope of this document is to report on the final development of toolboxes integration connectors, as a result of the experiences and software artefacts obtained in the second development iteration, towards producing a scalable, fault-tolerant, communication-efficient framework for cross-streaming data management and integration.

A scalable, fault-tolerant platform for big data by collecting, integrating and processing streams of data, including contextual data, is one key objective and main requirement for the Track&Know project. The platform should be based on an open architecture system ensuring both scalability and interoperability as well as open software standards, including a privacy by design approach to ensure that privacy and ethical issues are respected. The platform should include ready-to-go integration connectors for the seamless homogenization of multiple, voluminous and heterogeneous data-in-motion and data-at-rest sources. Moreover, the platform with the efficient management of diverse data sources and the provision of the connectors (with a continuous system availability of 24 hours a day, 7 days a week) will be able to support analytics and efficient spatiotemporal and contextual query-answering, complex data operators and Visual mobility analytics.

The development of the High-level Architecture of the Track&Know platform by highlighting the Data Sources and Data Store, the Connectors and Communication Platform, the underlying Infrastructure and Toolboxes is presented in this document. The design and development of the Track&Know platform takes account of business and interoperability requirements that have been collected and reported within the WP1 activities, by also considering the data diversity, volume and availability, in terms of extremely large and complex collections, and the detailed use-case scenarios described in WP6 and specifically in D6.1.

In order to provide a high-quality communication framework for Track&Know project, the connectors and the platform have been developed and deployed in a distributed environment, to enable development that can take advantage of the clustered deployment, demonstrate scalability, allow performance tests, provide metrics and finally allow for the identification of issues and related corrective actions in a timely manner, well before the Pilots utilise the platform.

Table of Contents

Document Summary Information.....	1
Revision history (including peer reviewing & quality control).....	2
Disclaimer	2
Copyright message.....	2
Executive Summary	3
List of Figures.....	5
List of Tables.....	8
Glossary of terms and abbreviations used.....	9
1 Introduction.....	12
1.1 Mapping TRACK&KNOW Outputs	12
1.2 Deliverable Overview and Report Structure.....	13
2 Track & Know Architecture.....	14
2.1 Requirements for the architecture for the management of structured & unstructured data streams	14
2.2 High-level Architecture.....	16
2.2.1 Data Sources and Data Store.....	17
2.2.2 Connectors and the Communication Platform.....	18
2.2.3 Infrastructure, Toolboxes and Pilots.....	18
2.3 Platform Implementation Milestones	19
3 Technologies and useful concepts related to Track&Know Connectors implementation.....	20
3.1 Apache Kafka.....	20
3.2 Producers and Kafka Connect	22
4 Cluster Provisioning, Scalability and System Architecture	24
4.1 Cluster Administration and Monitoring.....	26
4.1.1 The Grafana Track&Know Cluster Overview	26
4.1.2 Kafka Manager.....	29
4.1.3 Kafka Connect UI	30
5 Track & Know Datasets and Connectors	32
5.1 VFI Data Connectors.....	33
5.1.1 Kafka Producer type Connector for the VFI historical data	33
5.1.2 Kafka Connect type Connector for the VFI live data.....	45
5.2 SIS Data Connectors	52
5.2.1 Kafka Connect type Connector for the SIS DATASET1 data	54
5.2.2 Kafka Connect type Connector for the SIS CRASH data	57
5.2.3 Kafka Connect type Connector for the SIS EVENTS data	61
5.2.4 Kafka Connect type Connector for the SIS POSITIONS data.....	64
5.2.5 Kafka Connect type Connector for the SIS VOUCHER data.....	67
5.2.6 Introducing SIS datasets to the Track&Know Platform.....	70
5.3 PAP Data Connectors.....	73
5.3.1 Producer type Connector for the PAP reconstructed journey data	73
5.3.2 Kafka Connect type Connector for the VFI/PAP Smartphone app live data	81
5.4 Other available Connectors	86
5.4.1 Introducing data by using the Rest Proxy	86
5.4.2 Connectors available in the Confluent Hub	86
6 Conclusions.....	87
7 References.....	88
8 Annex	92

List of Figures

Figure 2.1 The Track&Know Real Time Processing Flow.....	15
Figure 2.2 The Track&Know components mapped to the BDVA reference model (source: www.bdva.eu).....	16
Figure 2.3 Track&Know High level architecture	17
Figure 2.4 Track & Know Platform Implementation Milestones	19
Figure 3.1 Apache Kafka Cluster Diagram taken from [7].....	20
Figure 3.2 Sample 3-Broker Apache Kafka Cluster	21
Figure 3.3 The Kafka Connect API, taken from [10].....	23
Figure 4.1 Track&Know Platform Cloud Computing Nodes	24
Figure 4.2 The noVNC remote desktop.....	26
Figure 4.3 The Track&Know Cluster Overview Dashboard	28
Figure 4.4 Kafka Manager displaying Brokers	29
Figure 4.5 Kafka Manager displaying Topic details.....	30
Figure 4.6 Kafka Connect UI displaying Connect Cluster	31
Figure 4.7 Kafka Connect UI displaying Connectors	31
Figure 5.1 The Track&Know repository	33
Figure 5.2 VFI customer data folders.....	33
Figure 5.3 VFI data in CSV files	34
Figure 5.4 VFI file contents.....	34
Figure 5.5 Parallelisation of the VFI data loading.....	35
Figure 5.6 Building and assembling the VFI batch producer.....	36
Figure 5.7 Producer jar and properties file	36
Figure 5.8 Topic creation for VFI batch data	38
Figure 5.9 Topic for VFI in Kafka Manager	39
Figure 5.10 VFI batch connector startup command	39
Figure 5.11 VFI batch connector startup output	40
Figure 5.12 Incoming message rates for VFI batch data.....	41
Figure 5.13 Multiple producers startup	42
Figure 5.14 Incoming message rates with multiple producers	43
Figure 5.15 Multiple producers (experiment finished).....	44
Figure 5.16 Reading VFI batch data from topic	45
Figure 5.17 Sample of VFI batch data message.....	45

Figure 5.18 Building the VFI live data connector.....	46
Figure 5.19 Kafka Connect jars directory.....	46
Figure 5.20 Kafka Connect UI for starting connectors.....	47
Figure 5.21 Selecting a new Connector	47
Figure 5.22 Configuring a Kafka Connect Connector.....	48
Figure 5.23 VFI live data connectors running.....	49
Figure 5.24 Reading VFI live data messages.....	50
Figure 5.25 Sample of VFI live JSON data.....	50
Figure 5.26 Reading VFI live data messages (AVRO)	51
Figure 5.27 Sample of VFI live AVRO data.....	51
Figure 5.28 Schema Registry entry for VFI live data.....	52
Figure 5.29 Building SIS connectors.....	53
Figure 5.30 Kafka Connect jars directory with SIS connectors.....	54
Figure 5.31 SIS DATASET1 collection	54
Figure 5.32 Kafka Connect UI and running connectors	55
Figure 5.33 Selecting the DATASET1 connector	55
Figure 5.34 Configuring the DATASET1 connector	56
Figure 5.35 SIS CRASH collection	58
Figure 5.36 Selecting the CRASH connector.....	59
Figure 5.37 Configuring the CRASH connector.....	59
Figure 5.38 SIS EVENTS collection	61
Figure 5.39 Selecting the EVENTS connector	62
Figure 5.40 Configuring the EVENTS connector	62
Figure 5.41 SIS POSITIONS collection.....	64
Figure 5.42 Selecting the POSITIONS connector	65
Figure 5.43 Configuring the POSITIONS connector	65
Figure 5.44 SIS VOUCHER collection.....	67
Figure 5.45 Selecting the VOUCHER connector.....	68
Figure 5.46 Configuring the VOUCHER connector.....	68
Figure 5.47 Creating the SIS topics	70
Figure 5.48 SIS topics seen in the Kafka Manager.....	71
Figure 5.49 SIS Connectors running.....	71
Figure 5.50 SIS Connectors at work.....	71
Figure 5.51 SIS topic data sample.....	73
Figure 5.52 PAP journey data in JSON files	74

Figure 5.53 PAP journey data JSON sample	75
Figure 5.54 Building and assembling the PAP journey batch producer	76
Figure 5.55 Producer jar and properties file	76
Figure 5.56 Topic creation for VFI/PAP journey data	78
Figure 5.57 Topic for VFI/PAP data in Kafka Manager	79
Figure 5.58 PAP/VFI batch connector startup command	79
Figure 5.59 VFI/PAP journey connector startup output	80
Figure 5.60 Incoming messages and rates for PAP journeys	81
Figure 5.61 Building the pap-vfi-live connector	82
Figure 5.62 The pap-vfi connector dir.....	82
Figure 5.63 Selecting the PAP/VFI connector.....	83
Figure 5.64 Configuring the PAP/VFI connector	84

List of Tables

Table 1 Adherence to TRACK&KNOW's GA Deliverable & Tasks Descriptions	12
Table 2 System Specification per Node	25
Table 3 Properties and sample values for the VFI batch producer	37
Table 4 VFI batch data loading runs	44
Table 5 Configuration options for the VFI live data connector	48
Table 6 SIS DATASET1 connector configuration options	56
Table 7 SIS CRASH connector configuration options	60
Table 8 SIS EVENTS connector configuration options.....	63
Table 9 SIS POSITIONS connector configuration options.....	66
Table 10 SIS VOUCHER connector configuration options	69
Table 11 Properties and sample values for the VFI batch producer	77
Table 12 Configuration options for the PAP/VFI live data connector	84

Glossary of terms and abbreviations used

Abbreviation / Term	Description
API	Application Programmable Interface
AMQP	Advanced Message Queueing Protocol
BD	Big Data
BDA	Big Data Analytics
BDP	Big Data Processing
BMDI	Big Mobility Data Integrator
BMI	Body Mass Index
CCG	Clinical Commissioning Groups
CER	Complex Event Recognition
CLI	Command Line Interface
CPAP	Continuous Positive Airway Pressure
CPU	Central Processing Unit
CSV	Comma Separated Values
DoA	Description of Action
DB	Database
DNA	Did Not Attend
DST	Day-light Saving Time
DVLA	Driver and Vehicle Licensing Authority
ESS	Epworth Sleepiness Scale
EtC	Ethics Committee
ETL	Extract Transform Load
FTP	File Transfer Protocol
GPS	Global Positioning System
GUI	Graphical User Interface

HBASE	Hadoop Database
HDFS	Hadoop Distributed File System
HGV	Heavy Goods Vehicle
HTTPS	Hypertext Transfer Protocol Secure
IO	Input/output
IT	Information Technology
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extensions
JVM	Java Virtual Machine
KPI	Key Performance Indicators
MQTT	Message Queue Telemetry Transport
NFS	Network File System
NIST	National Institute of Standards and Technologies
ODI	Oxygen Desaturation Index
OSA	Obstructive Sleep Apnoea
PMB	Project Management Board
PR	Pulse Rate
RPM	Rotations Per Minute
SASL	Simple Authentication and Security Layer
SFTP	SSH File Transfer Protocol
SLA	Service Level Agreement
SQL	Structured Query Language
SSH	Secure Shell
SSHFS	SSH Filesystem
SSL	Secure Sockets Layer

STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
TLS	Transport Layer Security
URL	Universal Resource Locator
UTC	Coordinated Universal Time
UV	Ultraviolet
VA	Visual Analytics
VM	Virtual Machine
WP	Work Package

1 Introduction

This deliverable is under “Development of Toolboxes Integration Connectors” which represents Task 2.3 of the Project. The scope of this document is to provide a detailed description of the software components developed in order to facilitate the seamless integration of data sources to the Track&Know Platform. This deliverable is provided as a complementary source of information, which highlights and documents various aspects of the implemented software, its usage and its configuration.

Moreover, in this document the architecture, system design, deployment specifics, technologies and useful concepts related to the Track&Know Platform and the connectors’ implementation are also presented to assist in the overall description of the produced software artefacts.

1.1 Mapping TRACK&KNOW Outputs

The purpose of this section is to map TRACK&KNOW’s Grant Agreement commitments, both within the formal Deliverable and Task description, against the project’s respective outputs and work performed.

Table 1 Adherence to TRACK&KNOW’s GA Deliverable & Tasks Descriptions

TRACK&KNOW GA Component Title	TRACK&KNOW GA Component Outline	Respective Document Chapter(s)	Justification
DELIVERABLE			
D2.3 Development of Toolboxes Integration Connectors	<p>Prototypes of the first version of the Track&Know integration connectors (Toolboxes), based on requirements defined in WP1. The connectors will facilitate the seamless integration of internal and external data sources.</p>	Chapter 3, Chapter 5 (Sections 5.1, 5.2, 5.3, 5.4)	Presentation of the developed connectors which facilitate the integration of internal and external data sources, their design, setup, operation and informative screenshots of their operation.
TASKS			
Task 2.3 Development of Toolboxes Integration Connectors (INTRASOFT, CEL, CNR, FRHF, VFI)	<p>This task is responsible for the development of the Track&Know integration connectors, based on requirements defined in WP1. During this task, we will pay particular attention to the technical integration of results produced by WP3-WP5, as well as the interaction with WP6 to ensure the successful deployment of the system to use-case and stakeholders’ sites. The task has two iterations. The first iteration is between M01 and M12. It will contribute to the overall Track&Know</p>	Chapter 2, Chapter 3, Chapter 5 (Sections 5.1, 5.2, 5.3, 5.4)	The respective document sections present in detail the software components designed and developed which provide the functionality described in the task, taking in account during the process requirements and input from interactions with other WPs.

	<p>architecture through early development of integration connectors. The experiences and software artefacts obtained in this process will become valuable input for the second development iteration. The result of the first iteration will be delivered as interim development of Toolboxes integration connectors. The second design and development iteration is between M12 and M18. It aims at integrating research results and improving the maturity of the integral Track&Know architecture. Thus, in the second iteration we will focus on the integration and testing with WP3-WP6. The result of the second iteration will be delivered as the final development of toolboxes integration connectors. The result of the task will be documented in deliverable D2.3.</p>		
--	--	--	--

1.2 Deliverable Overview and Report Structure

This deliverable presents an insight to the Track&Know custom Integration Connectors which have been implemented according to requirements, discussions in face to face ad hoc and plenary meetings and numerous conference calls organised with involved partners. While this deliverable resides as part of WP2, WP1 Requirements Analysis findings (within Task 1.2 and Task 1.3) and WP6, Experiments Planning and Setup [1], have been extensively taken into account.

This report is structured in the following way:

- **Chapter 1:** Introduction (this section), outlining the report and how it relates to the project as a whole.
- **Chapter 2:** Track&Know Architecture, presenting the High-level Architecture of the Track&Know platform, implementation milestones and the approach in general.
- **Chapter 3:** Technologies and useful concepts related to the Track&Know connectors implementation, providing a description of the tools and technologies that the Track&Know Big Data Platform is onboarding and offering to the Project, towards a fault-tolerant communication-efficient big data processing framework.
- **Chapter 4:** System architecture of scalable, fault-tolerant communication-efficient framework, describing the deployment of the platform presented in a realistic, industry resembling setup.
- **Chapter 5:** Track&Know datasets and connectors, providing an overview of the data made available by partners together with detailed information about the VFI, SIS and PAP data connectors complemented by informative figures related to their usage, configuration and performance.
- **Chapter 6:** Conclusions

2 Track & Know Architecture

The following section aims to present the high-level architecture of the Track&Know platform by highlighting the Data Sources and Data Store, the Connectors and Communication Platform, the underlying Infrastructure, Toolboxes and Pilots. Information provided within this section will serve as a quick overview to the reader, highlighting some aspects of the platform and allowing better understanding of the Connectors functionality within the platform. Extensive information regarding the Track & Know platform can be found in the deliverable D2.1 [2]. The design of the Track&Know architecture takes in account business and interoperability requirements that have been collected and reported within WP1 activities. Important platform implementation milestones are also presented.

2.1 Requirements for the architecture for the management of structured & unstructured data streams

The tools and technologies employed in the Track&Know Platform are carefully selected with high performance and high availability in mind, in order to meet the requirements extracted from the DoW, D1.2 (Corporate Big data requirements), D1.3 (Interoperability requirements) and pilot use cases.

It should be noted that the technologies selected allow clustered, highly available deployments that in addition to that can scale horizontally to accommodate increased workloads. As far as the advancements in interoperability are concerned, the aim is to achieve as many data flows as possible via Topics in a Publish/Subscribe fashion, which ensures that any data producer or consumer can provide or use data of the platform by simply abiding to the clearly documented approaches for reading from and writing to Kafka Topics.

It was necessary to adopt a scalable, fault-tolerant platform for big data to enable collection, integration and processing of streams and contextual data. This represented a key objective and main requirement for the Track-&Know project. The platform had to be based on an open architecture system which would ensure both scalability and interoperability as well as open software standards and include a privacy by design approach to ensure that privacy and ethical issues are respected. The platform also had to include ready-to-go integration connectors and allow the development of custom ones, for the seamless homogenization of multiple, voluminous and heterogeneous data-in-motion and data-at-rest sources.

It was also highlighted that the platform, with an efficient management of diverse data sources and the provision of the connectors (with a continuous system availability of 24 hours a day, 7 days a week) had to be able to support analytics and efficient spatiotemporal and contextual query-answering, complex data operators and Visual mobile analytics. Finally, the platform had to support the calculation, examination and reasoning upon relative performance indicators by obtaining measurements with respect to data consumption, rates, throughput, message count, derived data production etc. More detailed information on how the Track & Know platform fulfils the above and more requirements can be found in the deliverable D2.1 [2].

The needs and requirements originating from the above clearly necessitate a thorough approach that follows industry practices and can sufficiently provide adequate solutions for a high load of information exchange between interconnected systems and components, further allowing specific tuning depending on the (real-time, batch) processing needs emerging from the diverse project use-cases.

Moreover, accessing the available data sources should be straightforward enough to shift the focus of the researchers to the actual research pursuits of the project and not into how to connect to available data. The Platform should ideally allow the use of data sources by extending existing APIs and in a publish/subscribe fashion, while following at the same time approaches that are adopted in industry. This also helps towards a more extensive use of the produced tools by other domains.

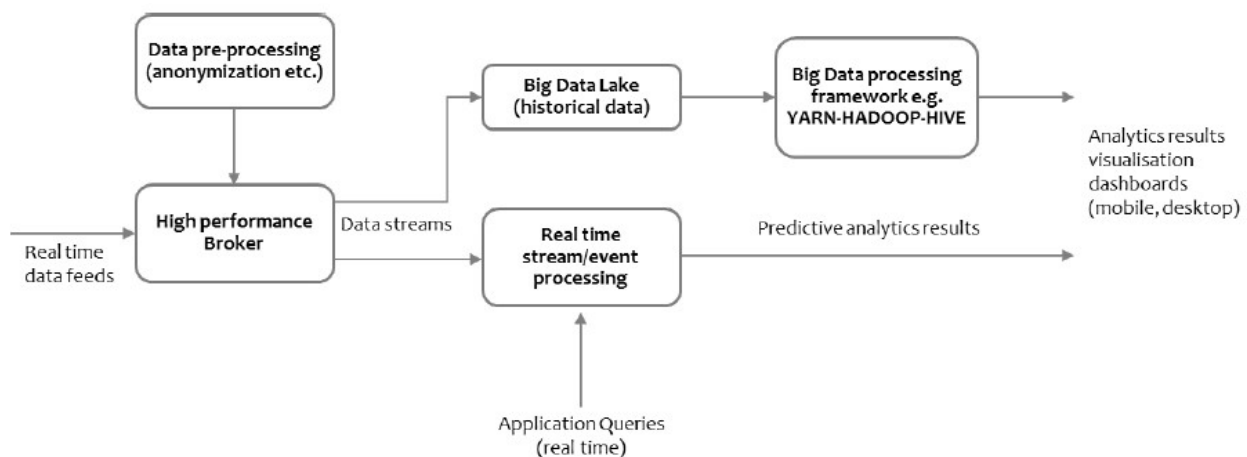


Figure 2.1 The Track&Know Real Time Processing Flow

Apache Kafka is characterized as a distributed streaming platform with, among other advantages and features, built-in capabilities for straightforward horizontal scalability. Kafka organizes data in partitioned topics and makes possible the

- i) Simultaneous processing of data that are entering the system from multiple clients with
- ii) Clients being able to read the incoming information at different offsets per client and with
- iii) The ability for clients to repeatedly process (replay) the streamed data.

Kafka can be configured for high throughput or tuned for low latency, depending on the case and is used successfully for real-time data streaming in the Industry [3].

When employing the described technologies, the business logic and solutions are implemented by incorporating distinct, loosely coupled components, which communicate via Apache Kafka. This enables potential simultaneous access to all the data available to the system, in order to produce results, either in batch or real-time scenarios. The approach of formulating solutions around Apache Kafka mainly incorporates the setup of Big Data processing pipelines.

It should be noted that Kafka can also cope very well with small message sizes (10-1024 bytes) when compared to other brokers, and this is very important to the use-cases of Track&Know, which feature big numbers of small mobility data messages. Apart from the increased performance in message rates, there is also evidence that Apache Kafka scales better when compared to other approaches [4]. It is worth mentioning that while other approaches can guarantee the delivery of messages up to “at least once”, Kafka supports “exactly once” delivery semantics between producers and consumers of messages simply by enabling specific idempotence configurations [5]. This is a very important feature, as exactly once delivery is a very complex problem in distributed systems and having this addressed out-of-the-box would be enough for many users to justify the selection of Kafka, especially if someone considers the increased performance and scalability characteristics.

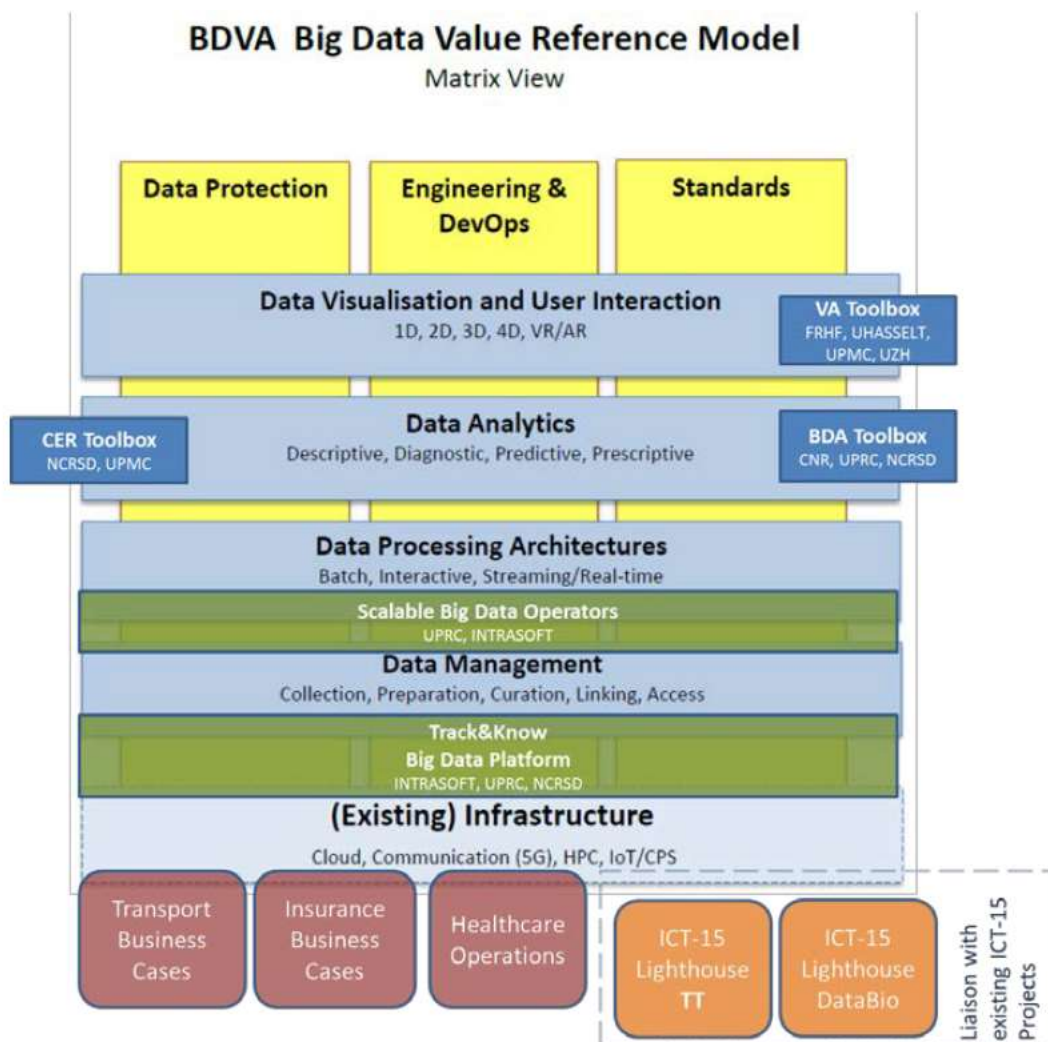


Figure 2.2 The Track&Know components mapped to the BDVA reference model (source: www.bdva.eu)

While Apache Kafka and specific tools that complete the platform are described further in the related deliverable D2.1 [2], it should be evident that the Track&Know Platform by utilizing the latter, is offering a more thorough approach, also promising to be adopted in other domains and industries. The Platform not only successfully fulfils the Data Management layer of BDVA shown in Figure 2.2 above, meeting Track & Know Objective 1, but also with the tools that the platform enables and onboards, a solid groundwork and even out of the box solutions are provided for developing Data Processing Architectures and facilitating the upward layers of the BDVA Model, providing thus an increased value of the Track & Know Platform not only for the specific project but for the community, other projects and other domains in general.

2.2 High-level Architecture

In this section the Track&Know High level architecture is presented, together with a description of the individual functional components, their interactions and the related workplan's Tasks. This architecture fulfils Big data requirements, described in D1.2, by also considering the data diversity, volume and availability in terms of extremely large and complex collections, and the detailed use-case scenarios described in WP6 and specifically in D6.1.

The architecture consists of:

- Data sources which represent the structured and unstructured data streams to be made available and be connected to the platform.
- Data store which represent the batch and interactive data sources that will be made available and will be connected to the platform.
- Connectors together with the Communication platform, that connect external Data sources and the Data store and make them available to the platform, Toolboxes and Pilots.
- Underlying Infrastructure providing all the necessary Big data tools.

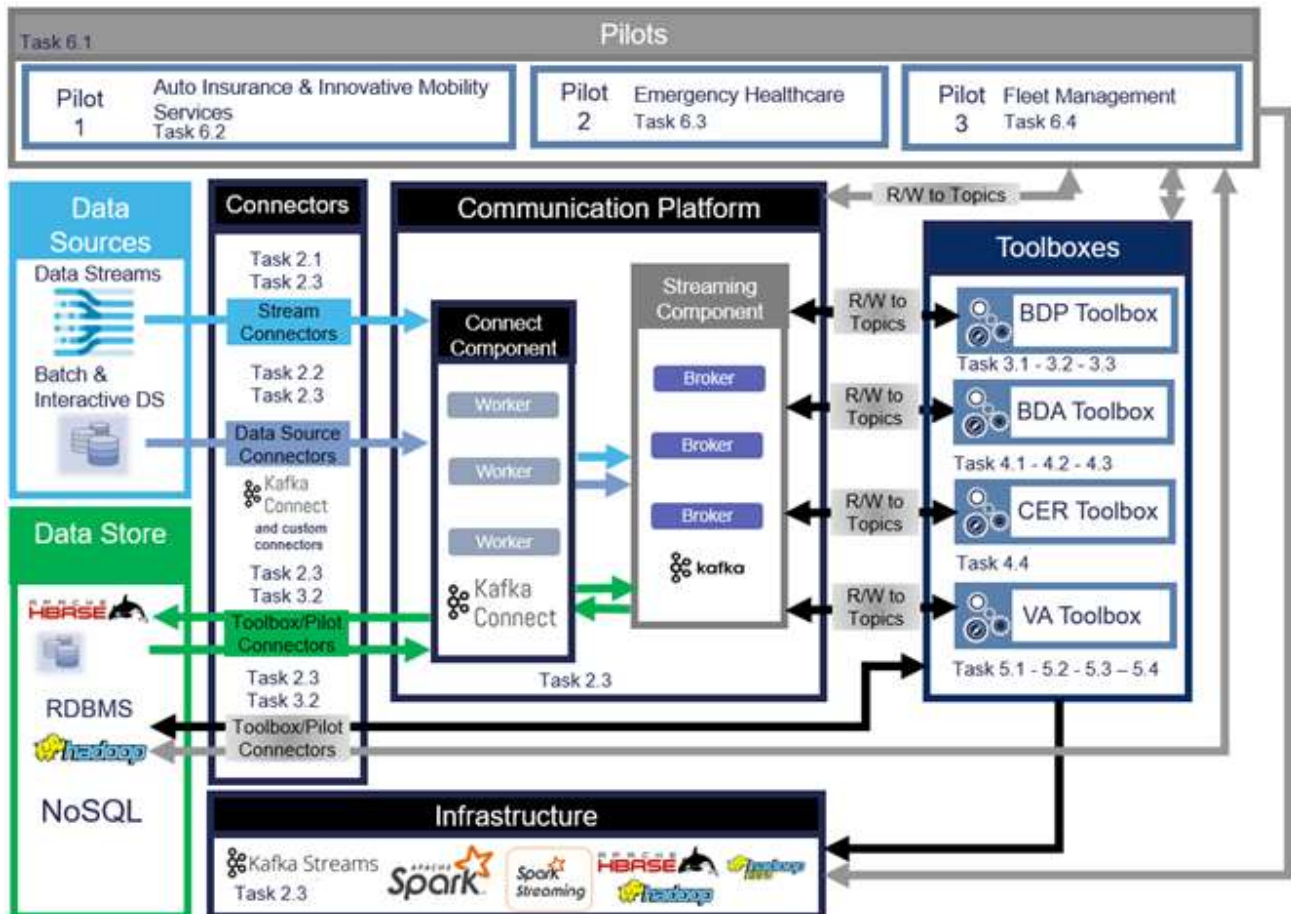


Figure 2.3 Track&Know High level architecture

2.2.1 Data Sources and Data Store

The external to the platform input data sources (Data Sources) represent the structured and unstructured data streams and the batch and interactive data sources that will be made available and will be connected to the platform. Similarly, and according to the requirements, appropriate persistent storage will be used, that can be seen in the input/output data components (Data Store). The Big data platform can efficiently interoperate with all the modern data storage technologies of a Big data ecosystem such as RDBMS, NoSQL, HDFS Hadoop, Apache HBASE, etc. as well as other persistence approaches such as Mongo, MySQL, JDBC, etc. The data sources are integrated and introduced to the platform by the means of integration connectors (Connectors) which will be described in this document.

2.2.2 Connectors and the Communication Platform

External data sources are connected and made available by employing the “Stream Connectors” and “Data Source Connectors”. These connectors are applicable to both historical and live data sources while operating in batch or real-time modes effectively introducing all the data available in Track&Know to the Platform for further processing and usage. The underlying technologies and capabilities of the Streaming Component and the multiple workers of the Connect Component enables the realisation of scalable and secure streaming data pipelines. The Communication Platform represents a distributed streaming platform, which is an Apache Kafka cluster. Apache Kafka allows publishing and subscribing to streams of records (Topics) similar to the functionality provided by a message queue, fully decoupling the components involved.

The streams of records are stored in a fault-tolerant durable way and consumers can process them as they occur, while multiple readers are allowed to read data in a topic at different offsets, replay the data, start from a specific offset, the beginning etc. In general, Kafka is suitable for building real-time streaming data pipelines to reliably exchange data between systems or applications and for building real-time streaming applications that transform or react to streams of data. The following Kafka APIs are available so that both Toolboxes and Pilots, shown in Figure 2.3 can interact with the Communication Platform according to needs. In general, an application would connect to a Broker to consume from, or produce to, one or more Topics (indicated above as R/W Read/Write to Topics):

1. “The Producer API allows an application to publish a stream of records to one or more Kafka topics”. [6]
2. “The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them”. [6]
3. “The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams”. [6]
4. “The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table”. [6]

For the development of the Integration Connectors, the Producer and Connector APIs have been extensively used during their implementation. In order to accommodate specific data needs of individual Toolboxes and Pilot use cases, the Connectors component includes purpose-built integration connectors that allow a Toolbox or Pilot module to access data by utilising the Kafka cluster. To avoid restrictions, it is always allowed to bypass the Kafka cluster and interact with the Data Store component of choice directly, by using Toolbox/Pilot Connectors. It should be noted that in this document the focus is on the Connectors that introduce data to the Apache Kafka Topics for further use within the processing pipelines.

2.2.3 Infrastructure, Toolboxes and Pilots

The underlying infrastructure consists of multiple Cloud VMs provided for each WP according to specific needs and hosts the necessary technologies required by the Toolboxes. It allows toolbox-specific storage and analysis of the data involved and the usage and execution of Toolboxes. It facilitates the execution of Pilots, by providing a distributed computing environment that supports the above, according to the technologies of choice for their realisation. As already described, the developed toolboxes (designed and developed within WP3, WP4 and WP5) are able to consume incoming data via the Communication Platform. The Toolbox code can connect to a Streaming Component Broker and consume data from a Topic. Similarly, the Toolbox code can write, persist and publish data to a Topic (that can be further used by e.g. another job, Toolbox or a Pilot). It is worth mentioning that the Communication Platform effectively provides access to all available Track&Know specific external data, introduced to it by the Connectors, via the Streaming Component and by using Topics. This approach allows the setup of batch and real-time stream processing applications also referred to as data processing pipelines.

Depending on the pilot use case, an individual Pilot interacts with and uses one or more Toolboxes in order to demonstrate a specific usage scenario. Therefore, a Pilot implementation indirectly accesses and processes incoming and existing data by using the Toolboxes. Furthermore, a Pilot implementation can consume derived data that the Toolboxes' processing produces. In the above architecture this is achieved as follows:

1. By writing derived data from a Toolbox to a Kafka Topic for the Pilot to consume. This utilises the Communication Platform and can allow a Pilot to consume derived data in real time.
2. By writing derived data from a Toolbox to the Data Store. The Pilot may or may not use the Communication Platform to access the derived data.

To satisfy Pilot data and processing requirements, the Toolbox/Pilot Connectors and the Communication Platform allows access to all data available in the system. The underlying infrastructure will enable execution of Pilot code and submitted jobs according to the requested technologies.

2.3 Platform Implementation Milestones

The implementation and deployment of the Track&Know Platform described in the previous sub-section included the development of the Track&Know integration connectors, according to requirements defined in WP1. The effort for the Track & Know platform realisation was assigned to WP2 of the Project. Throughout the process, technical integration of results produced by WP3-WP5, as well as the interaction with WP6 assisted the above tasks. The connector implementation task was split in two iterations, the first of which took place between M01 and M12, featuring an early development of integration connectors. At the time of writing, the initial development of connectors has been completed and the second iteration followed until M18 of the project. The Track&Know platform is successfully deployed and is fully operational in a cluster spanning multiple Cloud VMs. In addition to the above, the Consortium had the chance to attend a workshop specifically for accessing and using the Platform. Furthermore, a development, single-host stack of Apache Kafka and other tools, for experimentation and/or development tasks was provided, accompanied with instructions and guidelines. The initial experiences and software artefacts obtained in the first iteration served as input to the second iteration, which has fine-tuned the implementation of connectors in close collaboration with Toolboxes and Pilots. Integration and testing with WP3-WP6 resulted in a finalised version of the integration connectors.

In the timeline shown below, the two iterations mentioned together with important milestones are depicted:

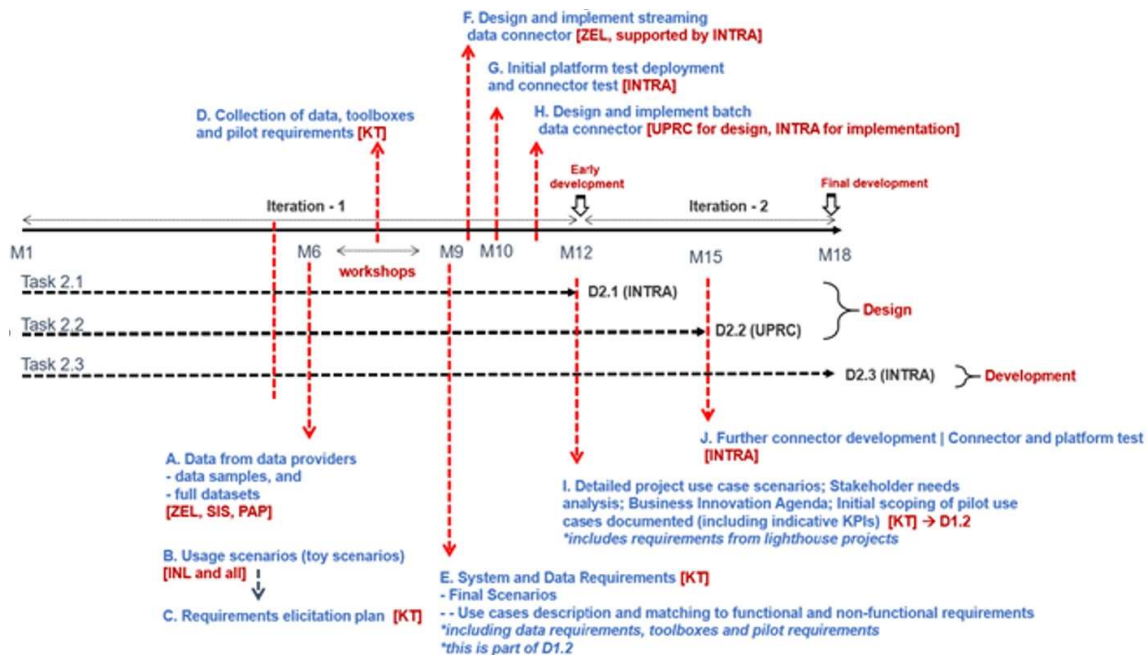


Figure 2.4 Track & Know Platform Implementation Milestones

3 Technologies and useful concepts related to Track&Know Connectors implementation

This section aims to provide a short description of the technologies and concepts that the Track&Know custom Connectors utilise, extend and are based upon. Apache Kafka, Kafka Topics, Kafka Producers and Consumers, as well as Kafka Connect are briefly presented to assist the reader in better understanding the produced Connectors. For a more detailed description regarding the Track&Know platform please consult deliverable D2.1 [2].

3.1 Apache Kafka

Apache Kafka is as an open-source, highly scalable, stream-processing platform maintained by the Apache Software Foundation. The software platform is designed to provide a high-throughput and low-latency ingestion, processing and consumption environment for real-time data feeds, that is usually run as a cluster on multiple hosts. Kafka offers four core APIs: Producer, Consumer, Streams and Connector, which are depicted in **Error! Reference source not found..**

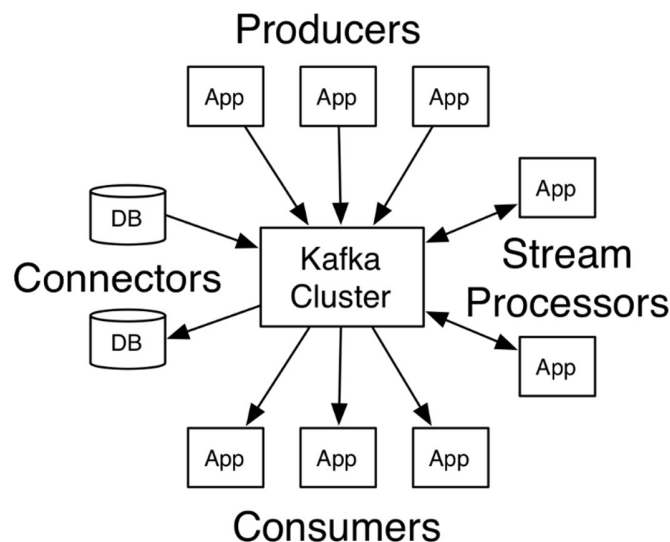


Figure 3.1 Apache Kafka Cluster Diagram taken from [7]

In an Apache Kafka Cluster it is usual to see three or more Kafka Brokers which provide a setup of adequate availability and replication for many scenarios. The Brokers themselves deal with the actual storage of the incoming data and provide the necessary functionality for producers and consumers to perform their work. Information in Kafka is stored in streams of messages or records that belong to a particular category called a "Topic". A Kafka record consists of a key, the actual value and a timestamp. While the actual value is provided by the data producer, the key may or may not be provided and the timestamp is added by Kafka.

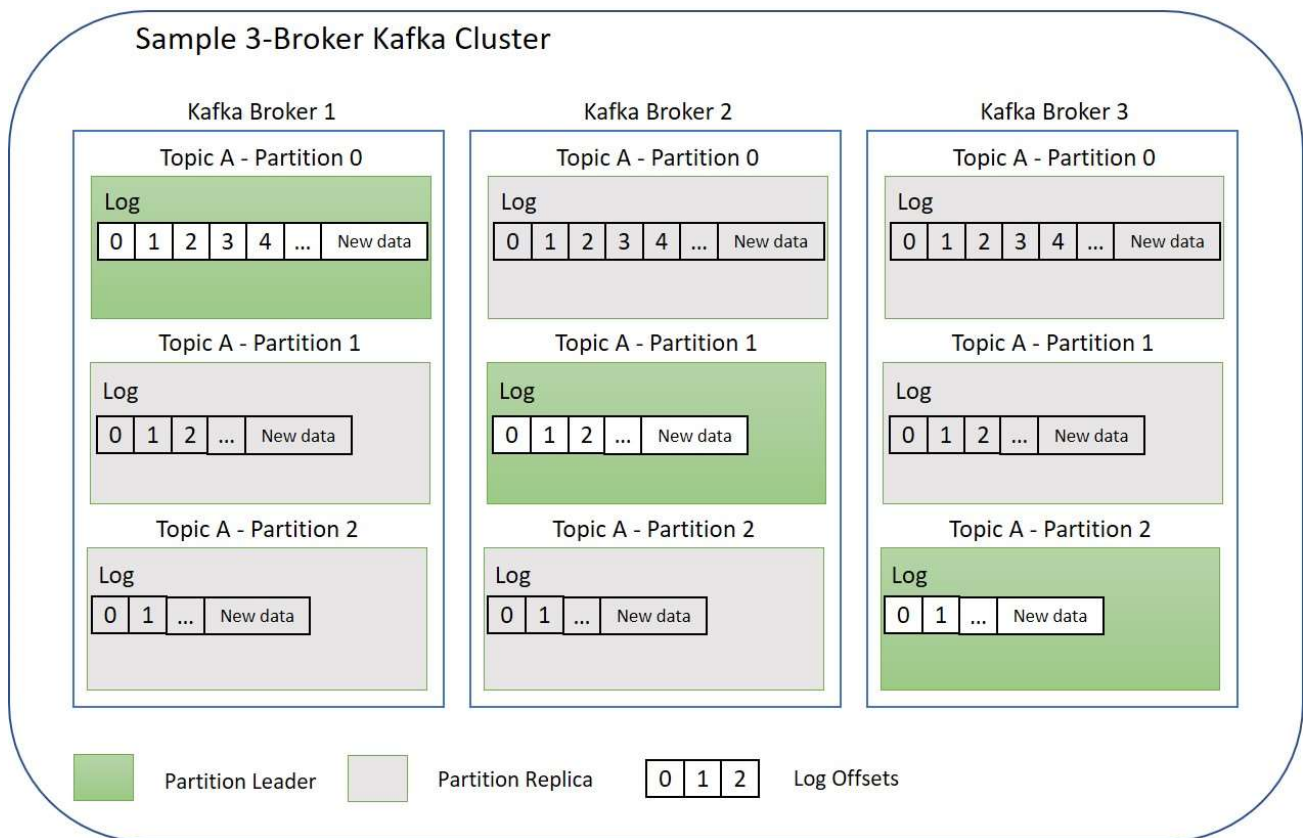


Figure 3.2 Sample 3-Broker Apache Kafka Cluster

Apache Kafka can achieve secure, encrypted communications between Brokers and Clients by utilising SSL (TLS). As far as compression of the exchanged information is concerned, Kafka offers end-to-end block compression. Data is compressed by the producer to be subsequently written as is (in the compressed format) on the Broker. When the data is to be consumed, the decompression occurs at the consumer end. Although Kafka can function as a message queue, it can offer a lot more functionality, such as allowing the persistence of a message after it has been read for further usage, operate even in cases where the data accumulated do not fit in the available memory and offer replication features not available in message queues. Kafka aims by design at persisting Big Data and increasing amounts of accumulated information without an impact to its performance. There are examples of Kafka clusters running in production with over a petabyte of stored data [8]. The duration for which data entering the cluster will be maintained and be available for further usage is configured on a per Topic basis and can last from hours to indefinitely. Kafka will automatically delete “expired” data to free up space in the cluster. An alternative approach allows the setting of a maximum amount of disk space that Kafka can take up and, in this setup, older data will be removed to accommodate new messages.

As Kafka stores information in a continuous log, it manages to maintain both information that has occurred while at the same time receiving information that is occurring, given that addition of new data to a Topic can go on indefinitely. If we consider a Kafka client that consumes information from a topic we can highlight that the same client that begins to read messages generated, for example, a year ago will continue to process the information until it reaches the current date and seamlessly proceed to process the data that are entering the system in real time as all the information is regarded as a continuous stream of data.

3.2 Producers and Kafka Connect

During the development of the Connectors the APIs provided which allow the efficient ingestion of data from various sources into the Track&Know Platform were extensively used. The Producer API and the Kafka Connect API are the ones that the Connectors extend and were built upon.

By extending the Producer API an application can publish a stream of records to one or more Kafka topics of interest. In general, a Producer sends messages to a Kafka Topic in key-value pairs. The key is used by the partitioner, which determines in which partition each record should be maintained. While there is an option to introduce a custom partitioner to control the way that the messages are spread among partitions based on some semantic partition function, Kafka provides the Default Partitioner, which is adequate in most cases. A typical producer would prepare the data and flush them to the Topic. Sending data is asynchronous and if required, the producer can receive a callback. The Producer API is well suited in cases where streams of data are to be transmitted, such as for example in IoT deployments, gathering of logs, loading batch data etc. When implementing a solution that extends the Producer API there is no limitation to the custom functionality that can be introduced in the Producer code which in turn enables elaborate ETL procedures to be implemented.

In a similar fashion, by extending the Consumer API an application can subscribe to one or more topics and process a stream of records that these topics contain and/or receive. The consumer can choose to read from the time of connection onwards or, if chosen, from the beginning of the Topic, receiving data that were produced to that topic prior to its connection. Although the Consumer API has been used extensively during the development of connectors to consume produced messages and test the incoming data, the main focus remains on the Producer and Connect APIs.

Kafka Connect is an open source framework which ships with Apache Kafka and is built on top of the Producer and Consumer APIs. Its purpose is to facilitate the high-performance streaming of data between Apache Kafka and other data systems in a scalable, reusable and reliable fashion. The simplifications and abstractions offered by the framework allow developers to quickly define and implement connectors that can move “large data sets into and out of Kafka” [9]. Uses of Kafka Connect among others include the ingestion of databases or the automatic capturing of updates in tables, the reading of files from several locations etc. into Kafka topics for further stream processing. Developers adhering to the Connect API are provided with all the necessary tools and groundwork that enables them to produce reusable, purpose-built connector plugins, which can be subsequently run in a Kafka Connect Cluster. The Connect Cluster which runs alongside the Kafka Brokers can be configured to distribute the tasks of a connector among the available workers of the Cluster for parallel processing, while at the same time providing a means of tracking the work performed and resuming the connectors in the case of failures with automatic offset management.

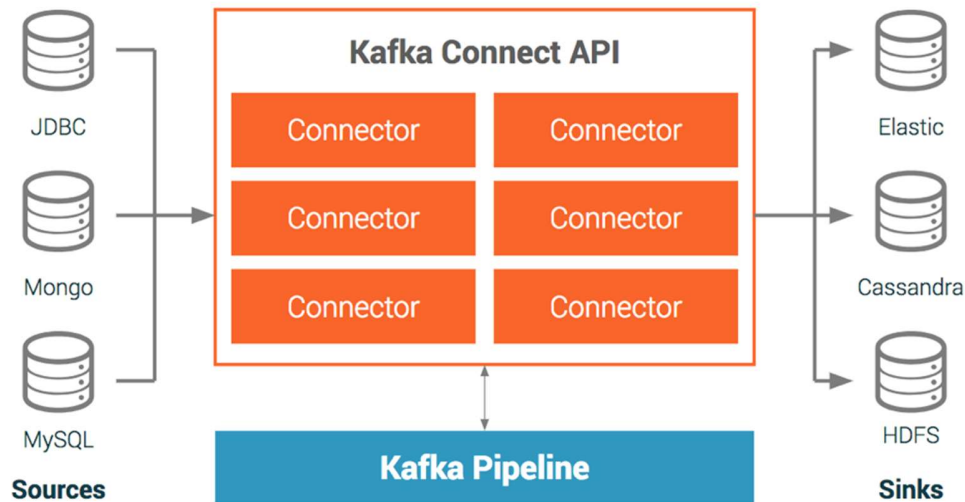


Figure 3.3 The Kafka Connect API, taken from [10]

Having Kafka Connect available in the Track&Know Platform also allows the adoption of readily available connectors which are in many cases certified and can significantly speed up the implementation of a solution by allowing the quick setup of data processing pipelines and high-performance data interconnections among the components of a system. In Track&Know this is not only a necessity for increased performance and incorporation of sound design approaches that will demonstrate the adoption of modern techniques in developing solutions, but also a relative KPI that aims to indicate a reduced time to realization for solutions. Furthermore, it is expected to assist the adoption of the Track&Know platform and tools to other domains as it will enable the introduction of other domain specific datasets with relative ease and assist in the interconnection of other system components, facilitating thus experimentation with other data sources.

All the custom connectors produced for Track & Know can be fine-tuned to achieve the desired performance characteristics, such as low latency or high throughput, or even trade off delivery and replication guarantees to achieve specific performance requirements.

4 Cluster Provisioning, Scalability and System Architecture

The Track&Know Platform is deployed in Hetzner Cloud [11] which provides the necessary flexibility of commissioning and decommissioning virtual machines according to current and future needs in the project. At the time of writing a total of 16 cloud computing instances of varying characteristics are operational, running Centos 7 Linux minimal installations with encrypted disk drives to ensure data encryption at rest. The login configurations for these machines do not permit root login and password authentication, allowing only key-based authentication via SSH.

Specific firewall rules fully restrict SSH traffic to selected IP addresses. Other traffic is in general only allowed between the cluster members, with the remaining IP addresses blocked by default, isolating the Platform from the outer world. Although the traffic between these hosts is routed internally by Hetzner, which allows for better network performance, the traffic still remains visible to a skilled attacker and therefore should be encrypted. At the time of writing, Hetzner Cloud is not offering a private subnet where the machines can be isolated from the rest of the Internet. In general, all data communications within the cluster are encrypted using SSL (TLS) by employing the encryption mechanisms offered by Apache Kafka. Furthermore, in cases where the data are served to other cluster nodes by an NFS server, then the alternative secure SSHFS is used. Finally, it should be mentioned that any remote desktop connections are performed via HTTPS and file uploads are using SFTP. The diagram below (Figure 4.1) provides an insight to the purpose of each node in the Track&Know Cluster.

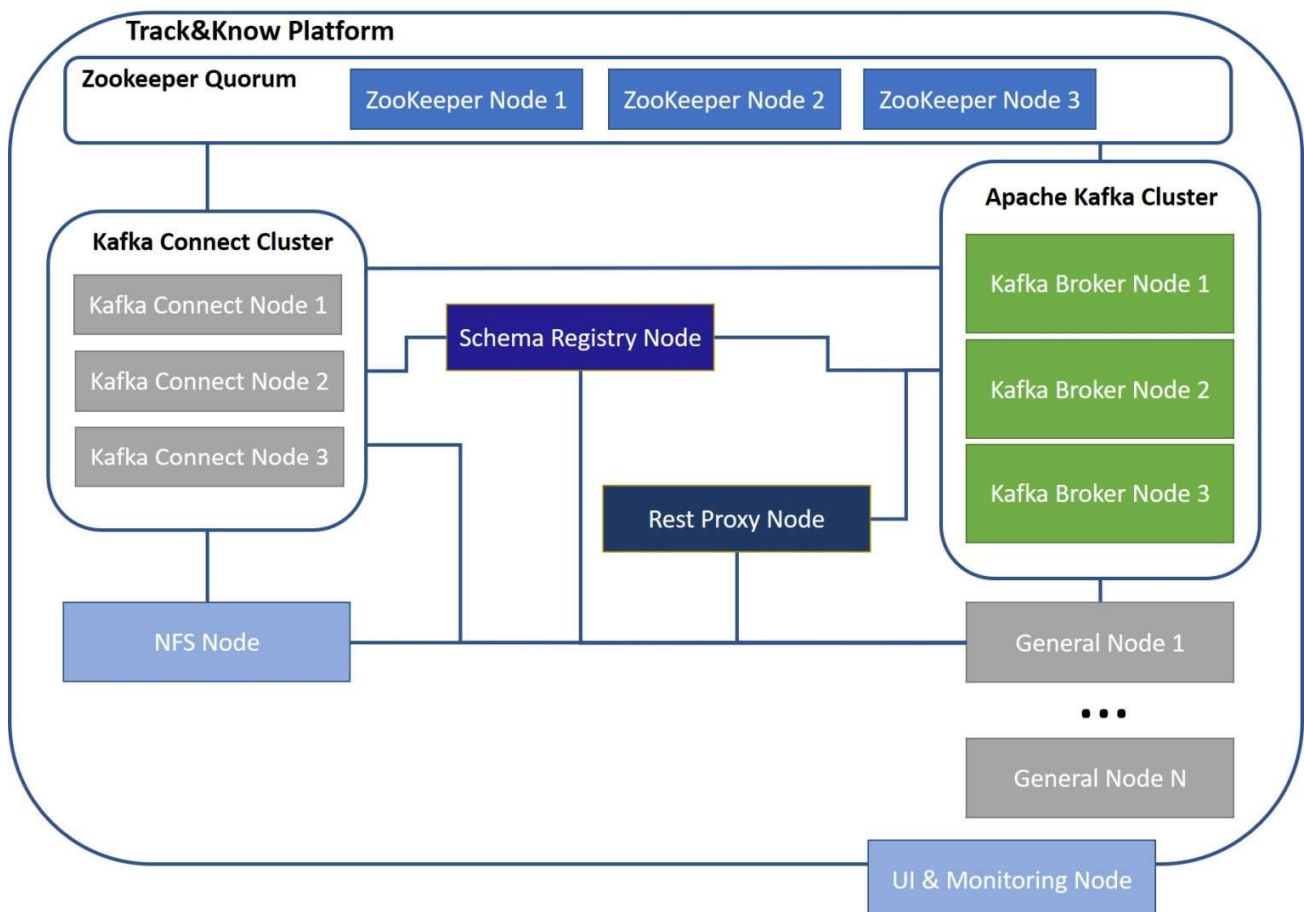


Figure 4.1 Track&Know Platform Cloud Computing Nodes

The Cloud computing provider hosts the configuration shown above that is adequate for the current needs of the Project and can be extended to accommodate future needs. Options include the introduction of new volumes to increase storage capabilities of nodes, the allocation of more vCPUs in cases that a node has increased computation needs and the increase of RAM for tasks that are memory intensive. While the configuration changes mentioned above are mainly related to Vertical Scalability, it should be made clear that they do not represent the main scalability approach for the Track&Know Communication Platform. The infinite, in principle, capacity of the Platform originates from the Horizontal Scalability characteristics, pertaining to the technologies selected and governs both their design and the design of solutions developed around them. To further highlight the characteristics of each of the cloud instances presented above, the following table is provided which describes the system specifications per node type.

Table 2 System Specification per Node

Node Type	CPU	RAM	SSD
Zookeeper Node	2 vCPU	4 GB	40 GB
Kafka Broker Node	8 vCPU	32 GB	240 GB
Kafka Connect Node	4 vCPU	16 GB	160 GB
NFS Node	4 vCPU	16 GB	160 GB
Rest Proxy Node	2 vCPU	8 GB	80 GB
Schema Registry Node	2 vCPU	8 GB	80 GB
UI & Monitoring Node	4 vCPU	16 GB	160 GB
General Node (WP3)	4 vCPU	16 GB	160 GB
General Node (WP4)	4 vCPU	16 GB	160 GB
General Node (WP5)	4 vCPU	16 GB	160 GB

Introducing more computing nodes is always possible by the Cloud computing provider, which can assume the roles that need to be supported in each case, also increasing fault tolerance. For instance, in Figure 4.1 it can be seen that the General Nodes, which are dedicated to specific WPs of the project can be increased depending on the requirements and needs of the toolboxes deployed. Similarly, in a case that the monitoring of Kafka Brokers indicates that the topics handled are loading the system, the Apache Kafka Cluster can be further extended by adding extra Brokers which will automatically spread the load among them allowing the increasing client needs to be handled. In general, the same approach holds for all the components shown in Figure 4.1 as the Kafka Connect nodes of the Connect Cluster, which are effectively workers executing the Connectors' tasks described in previous chapters, can be set up and added at will, with the same thing holding for the NFS nodes (more can be added), Rest Proxy and Schema Registry nodes (can even be run in a clustered manner).

The General Nodes appearing in Figure 4.1 represent computing nodes that are hosting a range of components of the Track&Know Project from simple Kafka Producers and Consumers, to Kafka Streams applications, Kafka KSQL Server(s), Toolboxes code and necessary technologies to support it etc. These nodes are provided so that

the solutions developed by other Toolbox WPs can be accommodated. It should be noted that these nodes are commissioned when a specific need arises, in order for Toolbox and supporting libraries to be configured. This also highlights the fact that the Cloud deployment of the platform is an active testbed for the Toolboxes early in the process, which provides feedback on how the solutions integrate and allows sufficient room for fine tuning. Taking in account the presented details, it becomes apparent that the Big Data solutions that the Track&Know Platform enables represent highly elastic, initially low-scale, entry-level approaches that can expand and contract according to the processing needs and the task in question. This translates to solutions that can be applied across many domains in ways that can be cost effective during periods of initial or low business, while remaining capable of growing for example when extra load is anticipated for a specific period, faster results should be obtained or simply in cases that processing needs are increasing.

4.1 Cluster Administration and Monitoring

For enabling good overview and easy access to the functionality offered by the several components of the Track&Know Platform, an extensive set of GUI components has been configured and are available for monitoring, administration and configuration usage. These are mainly open source components individually available in the community which have been selected and can be accessed when a user connects to a dedicated node which has UI capabilities (Figure 4.1 - UI & Monitoring Node, General WP Nodes). The connection to the remote desktop of that node is restricted to a specific list of allowed IP addresses, occurs strictly via HTTPS and is password protected. The user that wishes to access the remote desktop does not need to install any particular client due to the fact that noVNC [12] is used. Therefore, connecting to the remote desktop is as simple as opening an ordinary web page and does not require the user to open other ports locally or remotely as all traffic is routed via port 443 for HTTPS. When the browser window is maximised, the experience is very similar to working on the actual machine and has proven to work very well.

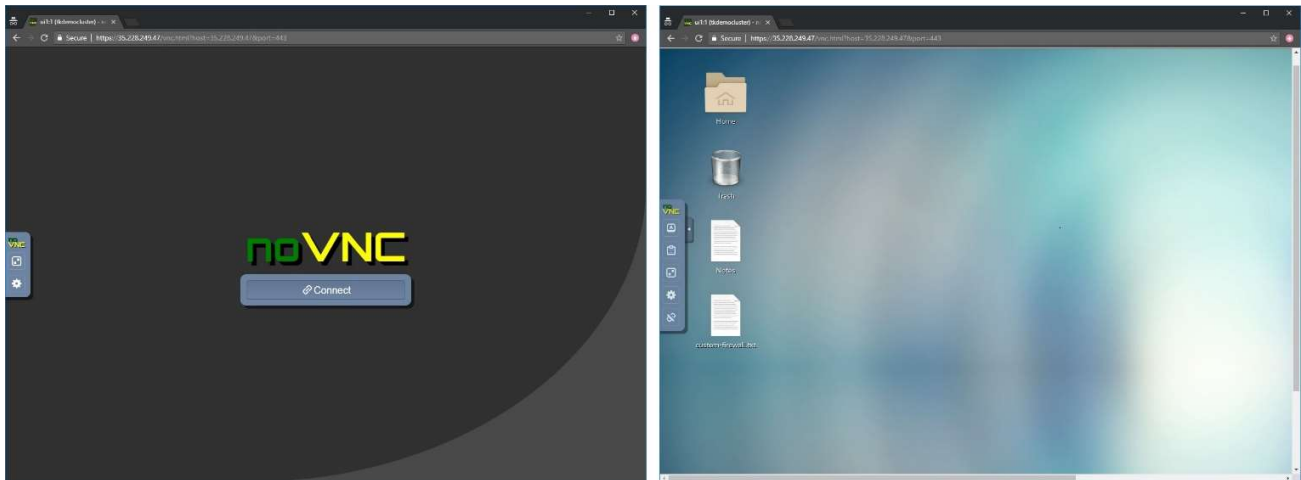


Figure 4.2 The noVNC remote desktop

4.1.1 The Grafana Track&Know Cluster Overview

The central monitoring approach adopted for the Track&Know platform utilises JMX exporters which are running on individual cluster nodes and expose a wide range of metrics depending on the node type. All these metrics are gathered at the UI & Monitoring Node where the Prometheus time series collection and processing server resides. All gathered data are available to compose informative dashboards which provide metrics, graphs and panels about the overall Platform status, performance and health.

The visualisation of the gathered metrics is achieved by a custom Track&Know Cluster Overview Dashboard seen in Figure 4.3. The current configuration allows the user to get a thorough overview of the Platform status with a single glance at the available metrics and graphs with the added ability to investigate historical performance

data. Currently Zookeeper, Apache Kafka, Schema Registry, Rest Proxy and Kafka Connect are monitored providing the following information:

- **Zookeeper:** Quorum Size, Alive Connections, Number of ZNodes, Number of Watchers.
- **Kafka:** Brokers Online, Active Controllers, Unclean Leader Election Rate, Online Partitions, Under Replicated Partitions, Offline Partitions Count, CPU Usage Per Broker (graph), JVM Memory Used Per Broker (graph), Time Spent in GC (graph), Messages In Per Topic (graph), Bytes In Per Topic (graph), Bytes Out Per Topic (graph), Messages In Per Broker (graph), Bytes In Per Broker (graph), Bytes Out Per Broker (graph).
- **Schema Registry:** Number of Kafka Schema Registry Servers, Active Connections, Open Rate, Close Rate.
- **Rest Proxy:** Number of Kafka Rest Servers, Active Connections, Open Rate, Close Rate.
- **Kafka Connect:** Number of Connectors, Connector Startup Success Total, Connector Startup Failure Total, Number of Tasks, Task Startup Success Total, Task Startup Failure Total, IO Rate per Connect Node (graph), Incoming Byte Rate per Connect Node (graph), Network IO Rate per Connect Node (graph).

The above information is very helpful when administering the Platform and monitoring Connector and Client performance and is a valuable tool during development and performance tests.

D2.3 Development of Toolboxes Integration Connectors

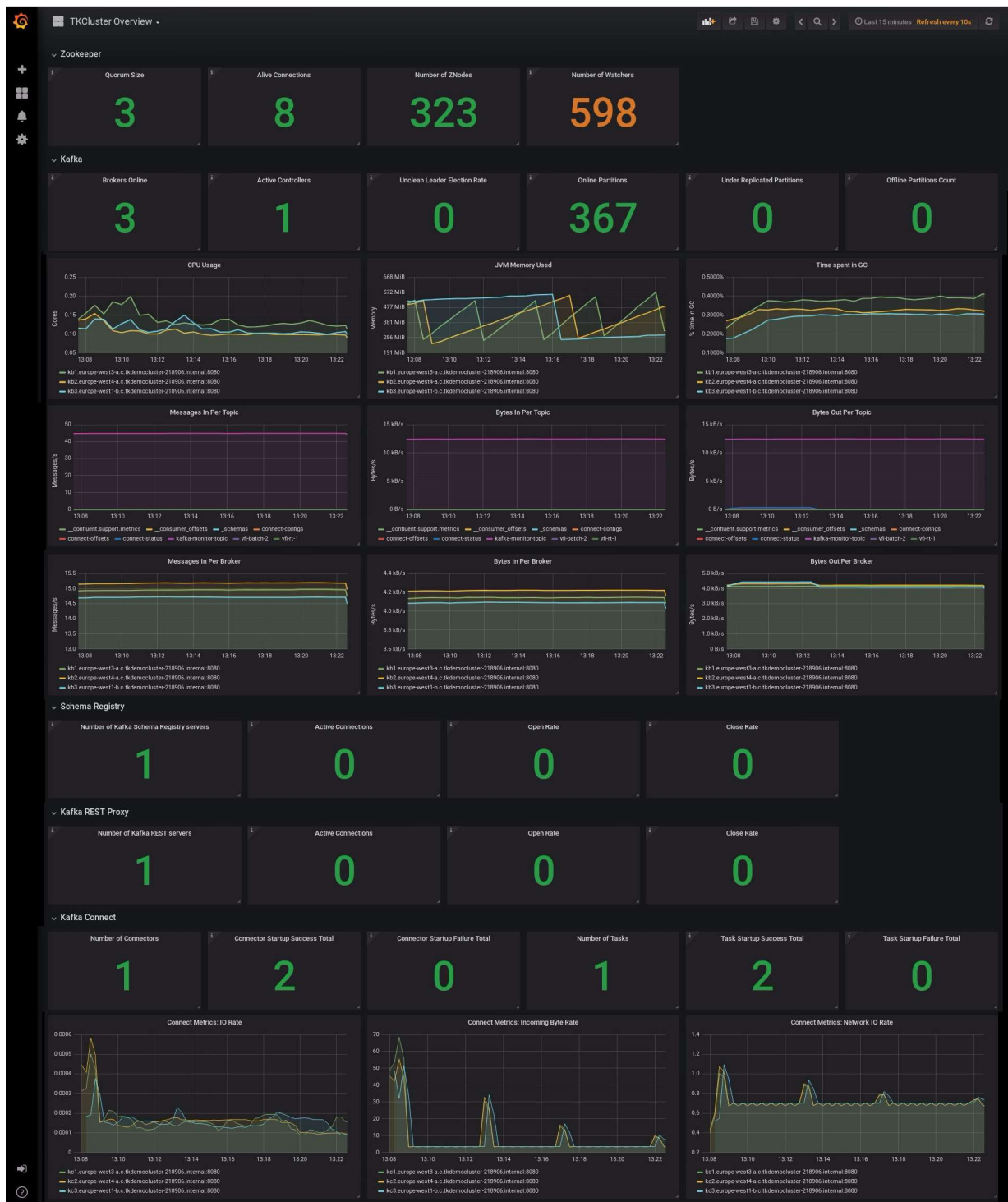


Figure 4.3 The Track&Know Cluster Overview Dashboard

4.1.2 Kafka Manager

Another tool that is provided with the Track&Know Platform, depicted in the following figures, is an open source tool developed by Yahoo for managing Apache Kafka Clusters. Kafka Manager [13] enables the user to inspect interesting aspects of the cluster including the number and state of Brokers, name and partitions of existing topics with number of messages and information regarding the way that individual partitions of topics are distributed among the cluster brokers. Furthermore, it allows a user to perform various administration tasks including listing of topics, creation of new topics with the ability to specify the number of partitions and replication factor, deletion of topics etc. Kafka Manager has been configured to work with the Track&Know Platform and is available to assist in administration and development tasks.

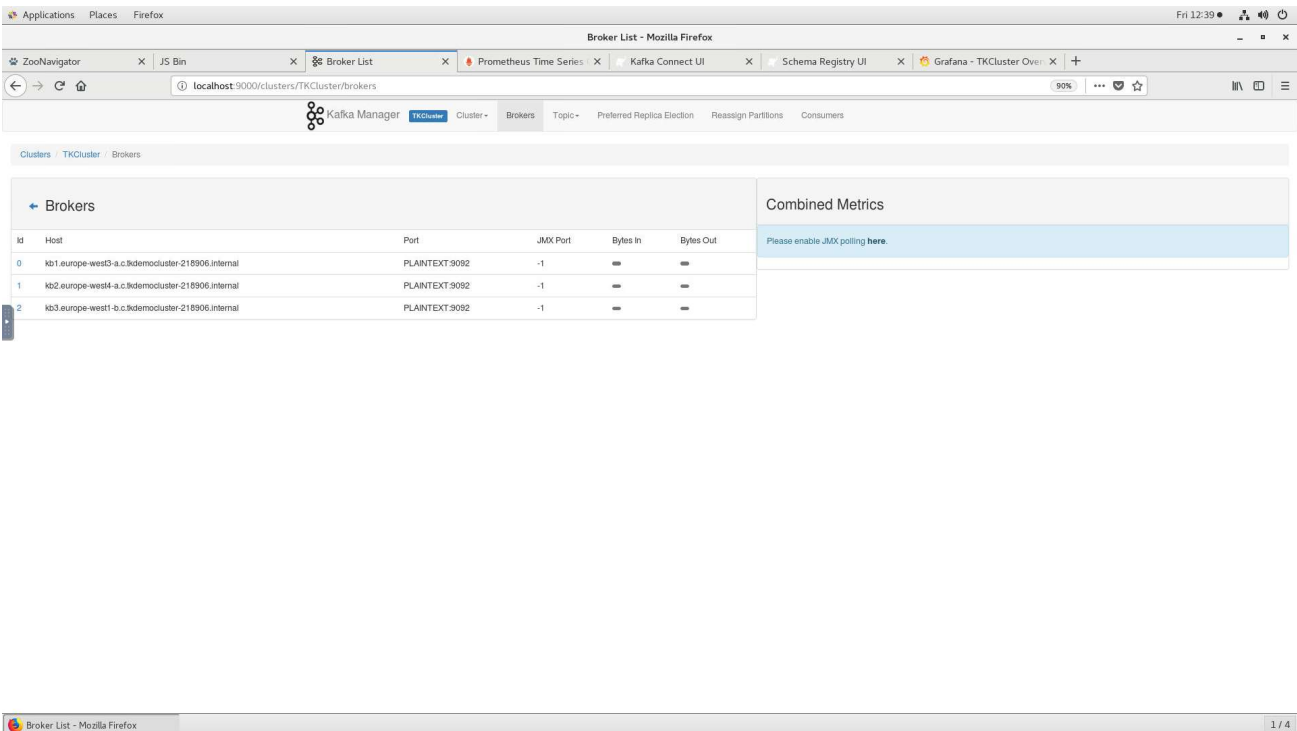


Figure 4.4 Kafka Manager displaying Brokers

D2.3 Development of Toolboxes Integration Connectors

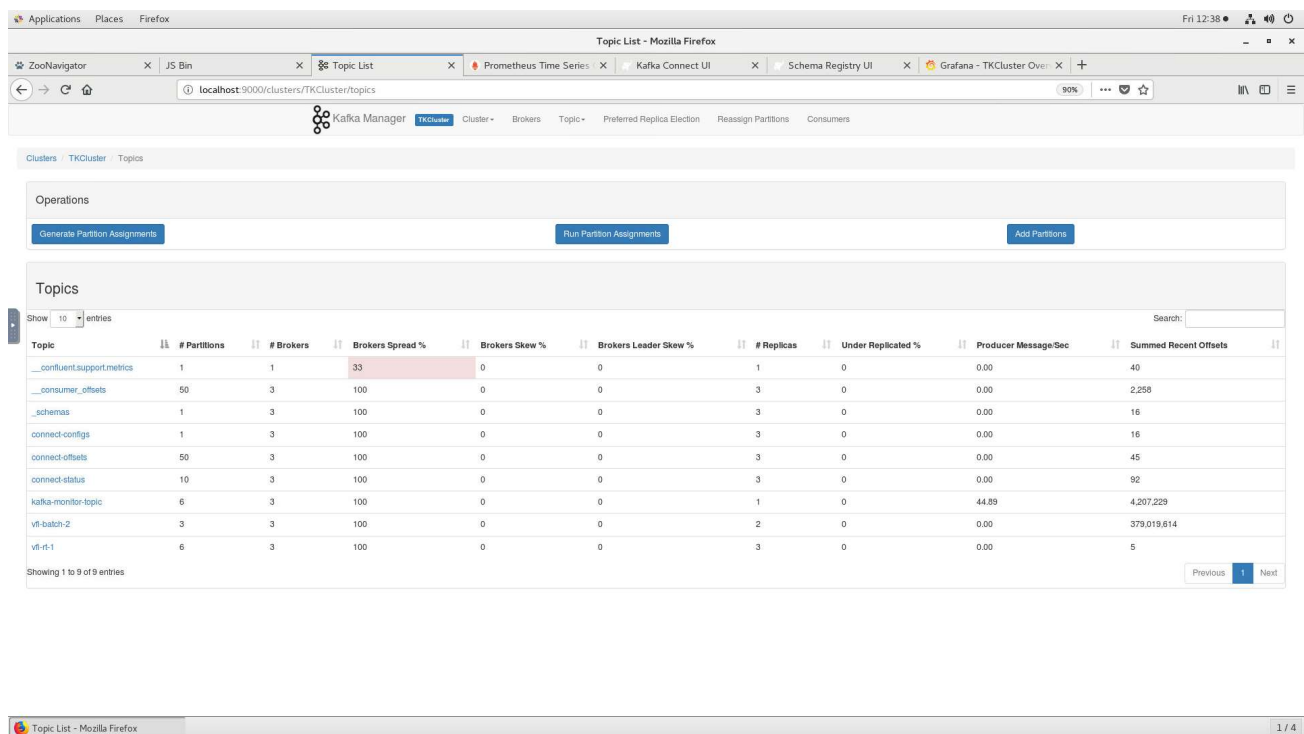


Figure 4.5 Kafka Manager displaying Topic details

4.1.3 Kafka Connect UI

The Kafka Connect UI [14] tool is a web-based interface that allows a user to setup, manage and gain an overview of the existing connectors in a Kafka Connect cluster. This open source tool can display the nodes that resemble the cluster and the running connectors per node, together with an overview of the running topology that depicts the source and sink connectors and the related topics. A very helpful feature of the tool is the ability to list the installed connectors in the cluster, pick the connector of interest and after providing the necessary configuration, start the connector. This feature is available for existing community connectors and also for custom connectors and has already been used to configure start, pause and restart Track&Know custom connectors in the process of their initial development.

D2.3 Development of Toolboxes Integration Connectors

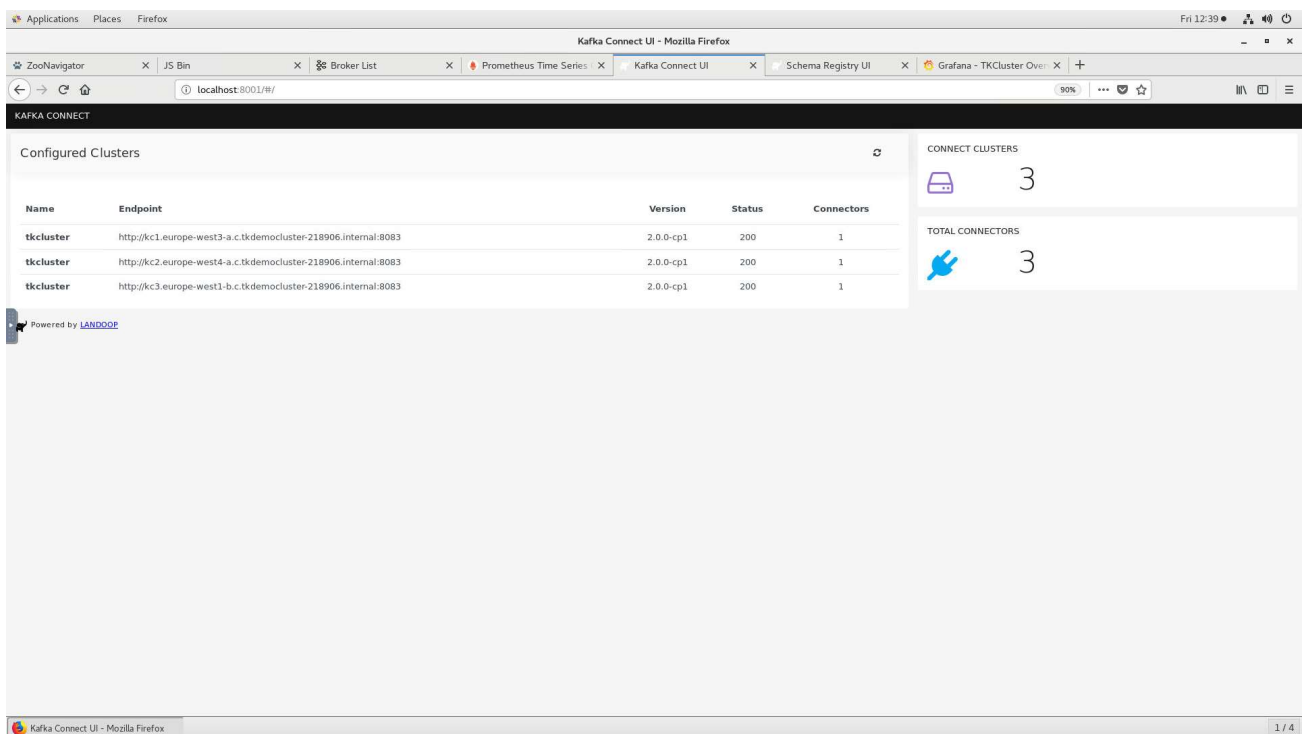


Figure 4.6 Kafka Connect UI displaying Connect Cluster

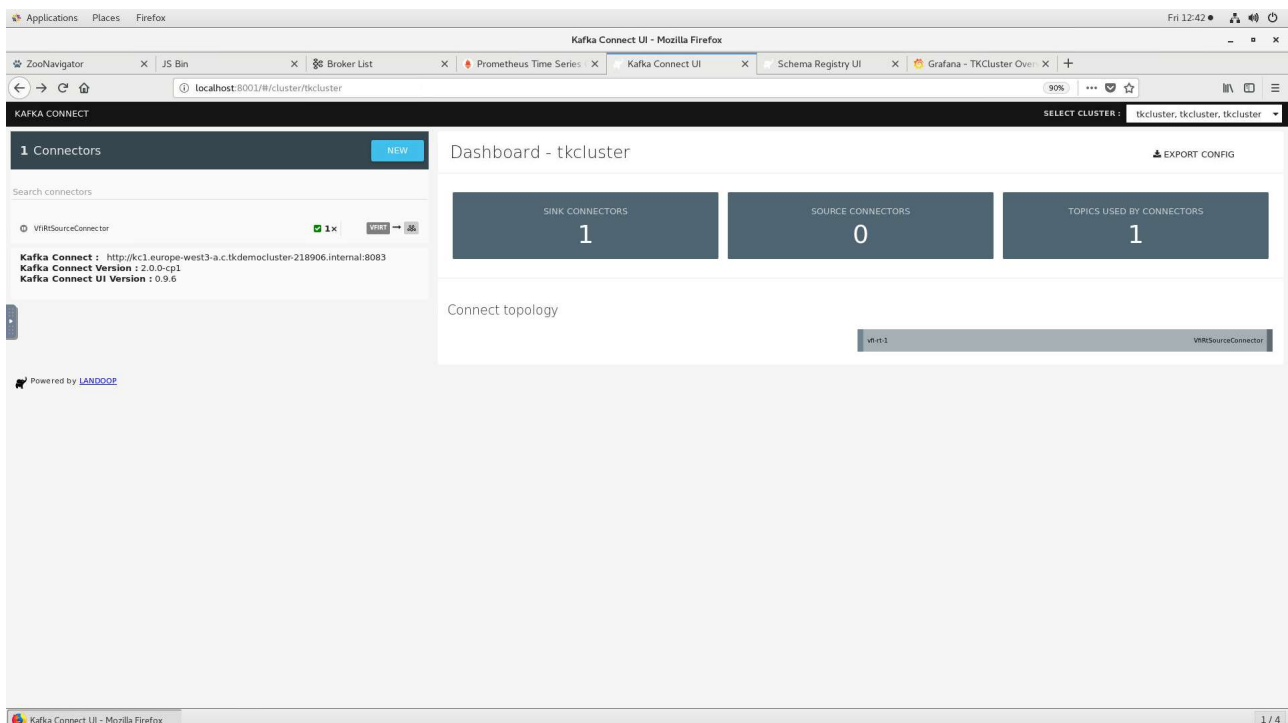


Figure 4.7 Kafka Connect UI displaying Connectors

5 Track & Know Datasets and Connectors

The available datasets within Track&Know originate mainly from partners VFI, SIS and PAP, detailed information for which can be found in deliverable D6.1 [1]. Partner VFI has provided historical fleet mobility data in CSV files organised in anonymised customer folders and has also provided a live data feed of mobility data for the vehicles that it monitors, via a REST API with the data in JSON format.

Partner SIS has provided access to a MongoDB instance, which was setup on their premises for accessing the data made available to the Consortium. The MongoDB contains a total of 5 collections split in 2 databases. Collection DATASET1 contains the main mobility data with relative GPS coordinates while collections CRASH and EVENTS contain accident and other type of important events respectively. The POSITIONS and VOUCHER collections of the database contain places of interest and insurance vouchers information. All the SIS data are stored as MongoDB documents and are retrieved in JSON format.

Regarding the data that partner PAP has made available, they consist of data related to the patients' journeys from residence to clinic, obtained by reconstructing GPS traces, directions, route timings and a poly-line as a GeoJSON for each individual appointment from existing appointment data. A plan also exists for introducing a purpose-built smartphone mobility data logger app developed by VFI, which provides patient journey information. It is planned for this data to be made available via a REST API in JSON format, in a similar way to the live feed of fleet mobility data described above. In cases where mobile networking costs should be avoided in a planned experiment, the app can delay transmission of accumulated data until WIFI is available. When operating in this mode the gathered data will be provided by using the same approach as for the VFI historical fleet mobility data. For the patient journeys data, a Producer type connector has been developed whereas for the logger app a Kafka Connect type of connector has been developed.

The data provided by the partners are loaded to the Track&Know platform by using a set of custom connectors implemented by using the Producer and Kafka Connect APIs. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later) and the source code is under version control residing in private Git projects/repositories hosted in GitLab.com:

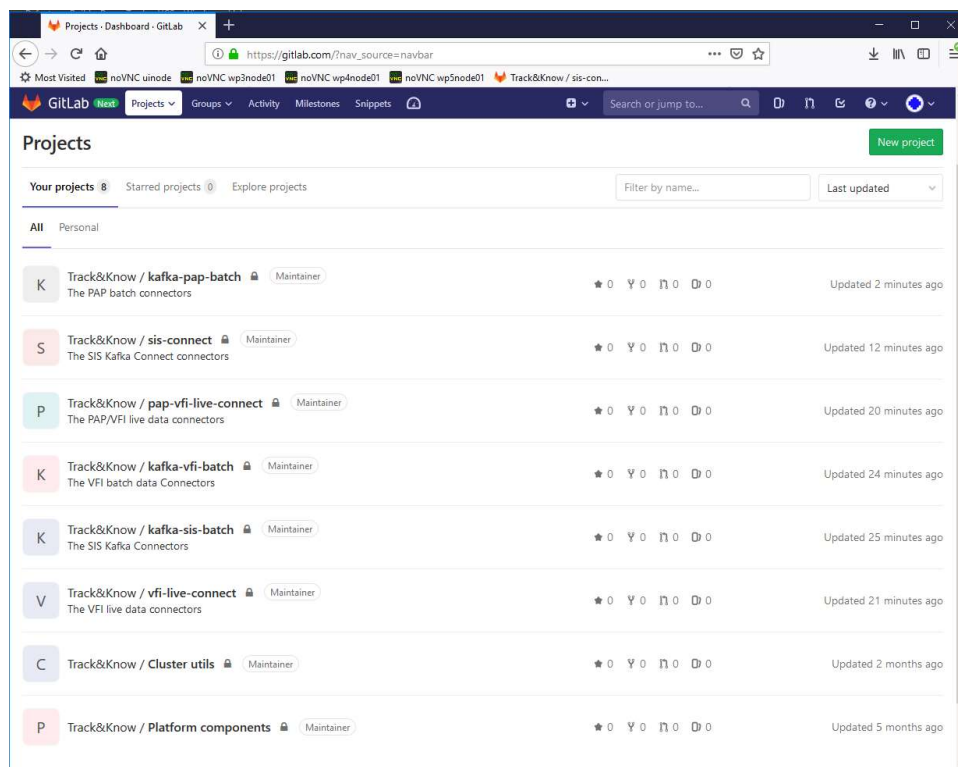


Figure 5.1 The Track&Know repository

The Connectors assume Apache Kafka V2.0 with Kafka Connect in a distributed deployment. In the Track&Know platform a 3-node Apache Kafka Brokers deployment is in place with an additional 3-node Kafka Connect Worker distributed setup. The deployment and usage of connectors is performed on virtual hosts running CentOS Linux (v7.6). It should be noted that all communications between the Connectors and the Track&Know Platform are encrypted (TLS v1.2) and compression is enabled using the Snappy algorithm.

5.1 VFI Data Connectors

In this section the connectors implemented for introducing the VFI data to the Track&Know platform are discussed, presenting information about their usage, configuration, startup and monitoring.

5.1.1 Kafka Producer type Connector for the VFI historical data

The dataset containing the VFI historical mobility data was made available in folders which contain multiple CSV files, with the latter holding the mobility data originating from the vehicles in question. Each folder represents an anonymised customer and the structure can be seen below:

```
[tkcluster@nfsnode01 sshfsshare]$ cd vfi-data/
[tkcluster@nfsnode01 vfi-data]$ ll
total 15184
drwxr-xr-x. 2 root root 16384 Feb 18 15:43 10200
drwxr-xr-x. 2 root root 8192 Feb 18 15:43 10230
drwxr-xr-x. 2 root root 8192 Feb 18 15:43 10260
drwxr-xr-x. 2 root root 4096 Feb 18 15:43 10290
drwxr-xr-x. 2 root root 4096 Feb 18 15:43 10320
drwxr-xr-x. 2 root root 12288 Feb 18 15:43 10710
drwxr-xr-x. 2 root root 4096 Feb 18 15:43 11040
drwxr-xr-x. 2 root root 8192 Feb 18 15:43 11250
```

Figure 5.2 VFI customer data folders

Several CSV files exist in every customer directory and are organised as follows:

```
[tkcluster@nfsnode01 sshfsshare]$ tree vfi-data/
vfi-data/
|-- 10200
|   |-- 3400_137530_2018-07-01-0000_2018-07-07-2359.csv
|   |-- 3400_137530_2018-07-08-0000_2018-07-14-2359.csv
|   |-- 3400_137530_2018-07-15-0000_2018-07-21-2359.csv
|   |-- 3400_137530_2018-07-22-0000_2018-07-28-2359.csv
|   |-- 3400_137530_2018-07-29-0000_2018-08-04-2359.csv
|   |-- 3400_137530_2018-08-05-0000_2018-08-11-2359.csv
|   |-- 3400_137530_2018-08-12-0000_2018-08-18-2359.csv
|   |-- 3400_137530_2018-08-19-0000_2018-08-25-2359.csv
|   |-- 3400_137530_2018-08-26-0000_2018-09-01-2359.csv
|   |-- 3400_137530_2018-09-02-0000_2018-09-08-2359.csv
|   |-- 3400_137530_2018-09-09-0000_2018-09-15-2359.csv
|   |-- 3400_137530_2018-09-16-0000_2018-09-22-2359.csv
|   |-- 3400_137530_2018-09-23-0000_2018-09-29-2359.csv
```

Figure 5.3 VFI data in CSV files

The data inside each one of the files are organised according to the following header:

company;vehicle;localDate;engineStatus;driver;driverEvent;longitude;latitude;altitude;angle;speed;odometer;satellites;fuelLevelIt;countryCode;rpm;levelType;fuelTankSize;vehicleOdometer;fuelConsumed;engineHours;closeToGasStation;deviceType;VehicleType;fuelRawValue;

A sample of the data inside a file can be seen below:

```
[tkcluster@nfsnode01 10200]$ less 3400_137530_2018-07-01-0000_2018-07-07-2359.csv
10200;3400_137530;2018-07-01 00:42:38.000;3;;23.713002;38.028115;80;196;0;0;20;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 01:42:38.000;3;;23.712997;38.028115;79;196;0;0;19;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 02:42:38.000;3;;23.712997;38.028115;79;196;0;0;15;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 03:42:38.000;3;;23.712980;38.028103;76;196;0;0;13;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 04:42:38.000;3;;23.712980;38.028103;76;196;0;0;12;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 05:42:38.000;3;;23.712980;38.028103;76;196;0;0;16;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 06:42:38.000;3;;23.712980;38.028103;76;196;0;0;17;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 07:42:38.000;3;;23.712980;38.028103;76;196;0;0;16;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 08:42:38.000;3;;23.712980;38.028103;76;196;0;0;17;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 09:42:38.000;3;;23.712980;38.028103;76;196;0;0;18;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 10:42:38.000;3;;23.713033;38.028188;90;196;0;0;15;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
10200;3400_137530;2018-07-01 11:42:38.000;3;;23.713030;38.028088;84;196;0;0;14;30;;CANBUS;0;;;0;FM5300;Truck 3,5 t;;
```

Figure 5.4 VFI file contents

Taking in account the above, it was decided to create a connector that could (if needed) upload the data in parallel. The approach that was adopted resulted in a Producer that can be run in several hosts at the same time and utilise multiple threads per instance for the reading and writing of the messages to the Apache Kafka Cluster.

When started, the Producer checks its configuration file which indicates the parent directory in which the customer folders are located. In the case that a single Producer instance is configured, it will proceed to ingest all the data by traversing each customer folder. If more instances are configured (e.g. 5) then the total customer folders number is divided accordingly, and a subset of them is assigned to each instance, effectively spreading the task among them. The separate instances of the Producer only make sense when started on a different host. For this reason, the Customer folders including all the CSV files are located on a dedicated node with the parent folder mounted via SSHFS on each and every other host where the Producer is run. The setup can be seen in the diagram below and it serves the purpose of making available the data to all the other nodes running Producer instances:

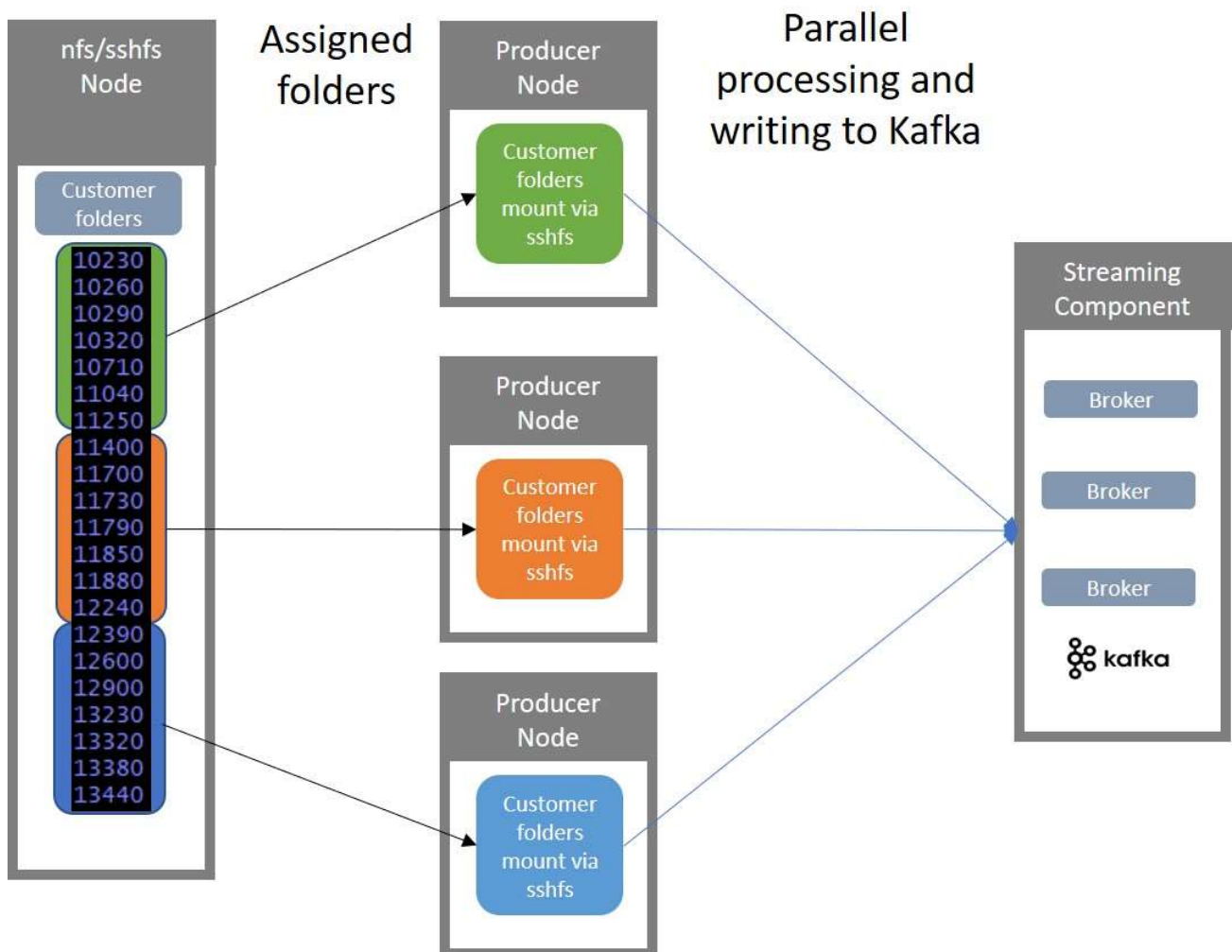


Figure 5.5 Parallelisation of the VFI data loading

As mentioned above, the multiple threads that can be configured on each Producer instance allow the maximum CPU utilisation of each Producer node, maximising the throughput. The Producer threads are equally assigned between them several customer folders from the subset that is assigned to this Producer instance.

Once the Producer instances are started, the individual threads begin to read customer folders in parallel. The files inside each customer folder are sorted in a chronological order according to their filename and are one by one loaded and parsed. Each line results in a message that is sent to the Kafka Topic of choice. If the destination Kafka Topic has more than one partition configured, then the messages are introduced to the applicable partition based on the customer number.

Due to the fact that a Customer folder is assigned to only one Producer instance and also because this folder will be processed entirely by a single thread of this Producer instance, the ordering of messages will be preserved within the destination partition of the Topic. This guarantees that the data for a particular customer will be written in the order that they are read from the CSV files. Because multiple threads exist which may write other customer data to the same partition, there is interleaving of customer messages within a specific partition but the order of messages of a specific customer is always preserved. Subsequent enrichment tasks have been found to work fine with the above and can parallelise in a way that is similar to what has been presented.

The connector implementation also includes a small cleaning feature which proved to be necessary as the provided CSV files were found in some cases to include the Unicode Byte order mark (BOM - Character U+FEFF) that is removed. Furthermore, another version of the connector was implemented that is capable of discarding messages that contain GPS coordinates outside the Europe rectangle prior to loading them to the Kafka Topic. Although it was found that performance was not impacted, it was chosen to assign this functionality to WP3 developed toolboxes as it was decided that it better fits their purpose.

The Connector is packaged in a jar and can be built from the sources by using mvn clean, compile and assembly: single as it can be seen below (IntelliJ IDE):

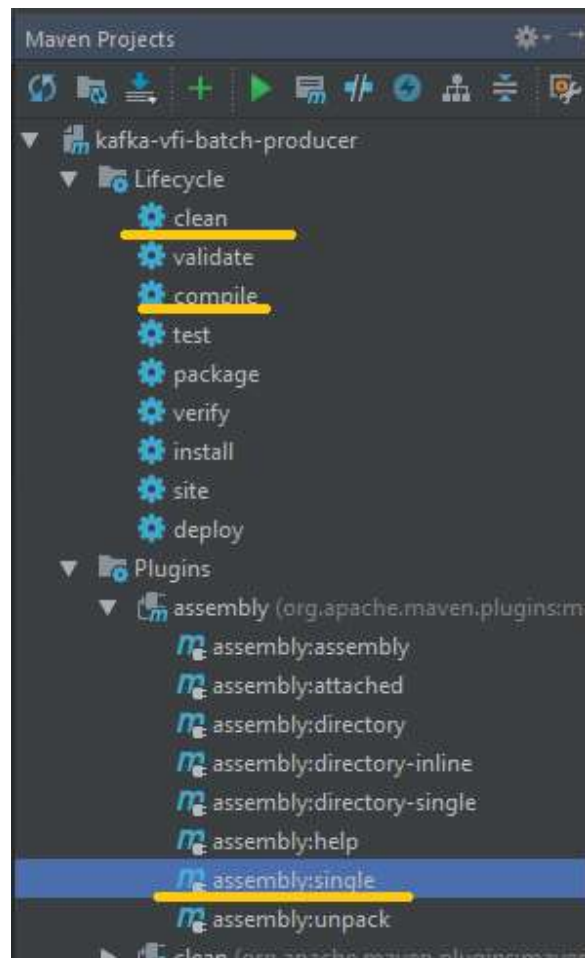


Figure 5.6 Building and assembling the VFI batch producer

The jar that is produced can then be copied to the VMs that the Connector will be run. Alternatively, the code can be pulled at a VM and the connector can be built there. Once the jar is built it should be made executable (sudo chmod +x) and placed in a folder of choice together with its property file:

```
[tkcluster@nfsnode01 vfi-batch-producer]$ ll
total 7564
-rwxr-xr-x. 1 tkcluster tkcluster 7738606 May  9 16:35 kafka-vfi-batch-producer-1.0-jar-with-dependencies.jar
-rw-rw-r--. 1 tkcluster tkcluster   1211 May  9 16:35 vfiBatchproducer.properties
[tkcluster@nfsnode01 vfi-batch-producer]$
```

Figure 5.7 Producer jar and properties file

The property file must have the name vfiBatchproducer.properties and exist in the same folder as the jar, otherwise it cannot be read and the connector will fail to start. The Connector uses encryption (TLSv1.2) and

compression (Snappy algorithm) while sending data to Apache Kafka and offers configurable batching and exactly once delivery. These and other properties are configured in the properties file and are explained below in more detail (example shown):

Table 3 Properties and sample values for the VFI batch producer

Property and Sample value (Example)	Description
<code>work.with.dir=/home/wp3user04/vfi-sample-data</code>	The directory in which the batch connector will look for data. The data should be organised in the format provided by VFI i.e. client folders which contain csv files.
<code>topic=wp3-user04-test1</code>	The Kafka topic that this connector will write data to
<code>worker.threads.count=8</code>	The number of worker threads that this connector instance will employ when writing data
<code>producer.id=0</code>	The id of this producer. If there are multiple, then all must have different ids.
<code>total.producers=5</code>	The total number of producers. The loading task will be split among them and they will run in parallel.
<code>key.serializer=org.apache.kafka.common.serialization.StringSerializer</code>	The serializer used for the message key
<code>value.serializer=org.apache.kafka.common.serialization.StringSerializer</code>	The serializer used for the message value
<code>bootstrap.servers=static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093</code>	These are the 3 Kafka Brokers to which the Connector will send data. Note that port 9093 is used because only encrypted connections are allowed.
<code>security.protocol=SSL</code>	Authentication and encryption settings – Enables security.
<code>ssl.enabled.protocols=TLSv1.2</code>	Authentication and encryption settings – Enforces TLSv1.2
<code>ssl.truststore.location=/home/wp3user04/ssl/kafka.client.truststore.jks</code>	Authentication and encryption settings – Location of the Truststore that will be used to test Broker certificates.
<code>ssl.truststore.password=password</code>	Authentication and encryption settings – Truststore password

ssl.keystore.location= /home/wp3user04/ssl/kafka.client.keystore.jks	Authentication and encryption settings – Location of the Keystore where this Producer's certificate is stored.
ssl.keystore.password= password	Authentication and encryption settings – Keystore password
ssl.key.password= password	Authentication and encryption settings – Key password
enable.idempotence=true	Setting for idempotent producer. This will enable exactly once delivery.
acks=all	Setting required for idempotent producer. This will enable acks from all replicas.
retries=2147483647	Setting required for idempotent producer. This will enable max number of retries of failed messages.
max.in.flight.requests.per.connection=5	Setting required for idempotent producer. This will enable at most 5 unacknowledged messages while sending data.
compression.type=snappy	Compression algorithm used
linger.ms=5	Performance specific settings. Time to wait for a batch to fill. Optimal values may vary depending on cluster, message sizes etc.
batch.size=16384	Performance specific settings. Batch size. Optimal values may vary depending on cluster, message sizes etc.
request.timeout.ms=10000	Performance specific settings. Time to wait for a request to complete. Optimal values may vary depending on cluster, message sizes etc.

The above properties must be configured before the Connector can be started. Ensure that the topic is created beforehand and according to replication, partitioning, retention and size needs:

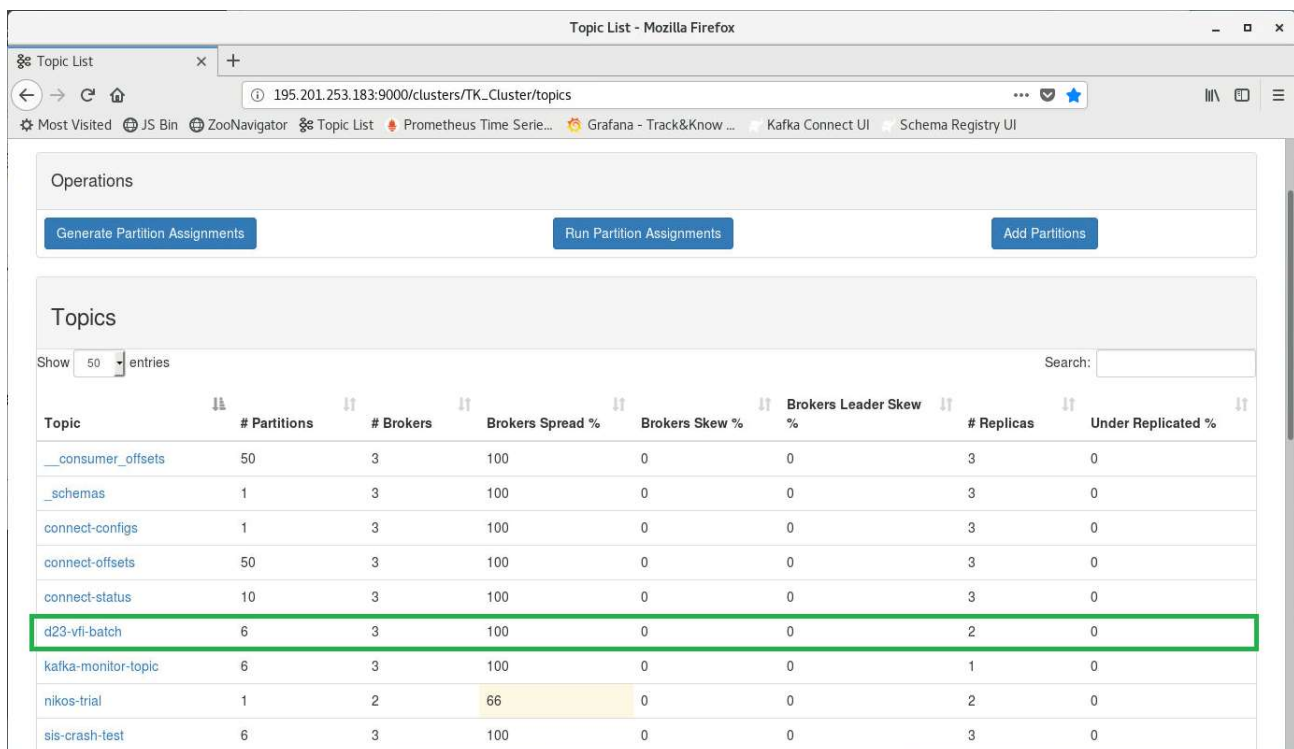
1. Please ensure that the topic where the data will be loaded has been created before starting the VFI Batch Producer Connector instances:

```
[wp3user04@wp3node01 ~]$ ~/confluent-5.0.0/bin/kafka-topics --create --zookeeper zookeeper.connect=195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-vfi-batch --replication-factor 2 --partitions 6
Created topic "d23-vfi-batch".
[wp3user04@wp3node01 ~]$ _
```

Figure 5.8 Topic creation for VFI batch data

The topic just created can be seen in Kafka Manager presented in earlier sections of the document:

D2.3 Development of Toolboxes Integration Connectors



Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %
__consumer_offsets	50	3	100	0	0	3	0
_schemas	1	3	100	0	0	3	0
connect-configs	1	3	100	0	0	3	0
connect-offsets	50	3	100	0	0	3	0
connect-status	10	3	100	0	0	3	0
d23-vfi-batch	6	3	100	0	0	2	0
kafka-monitor-topic	6	3	100	0	0	1	0
nikos-trial	1	2	66	0	0	2	0
sis-crash-test	6	3	100	0	0	3	0

Figure 5.9 Topic for VFI in Kafka Manager

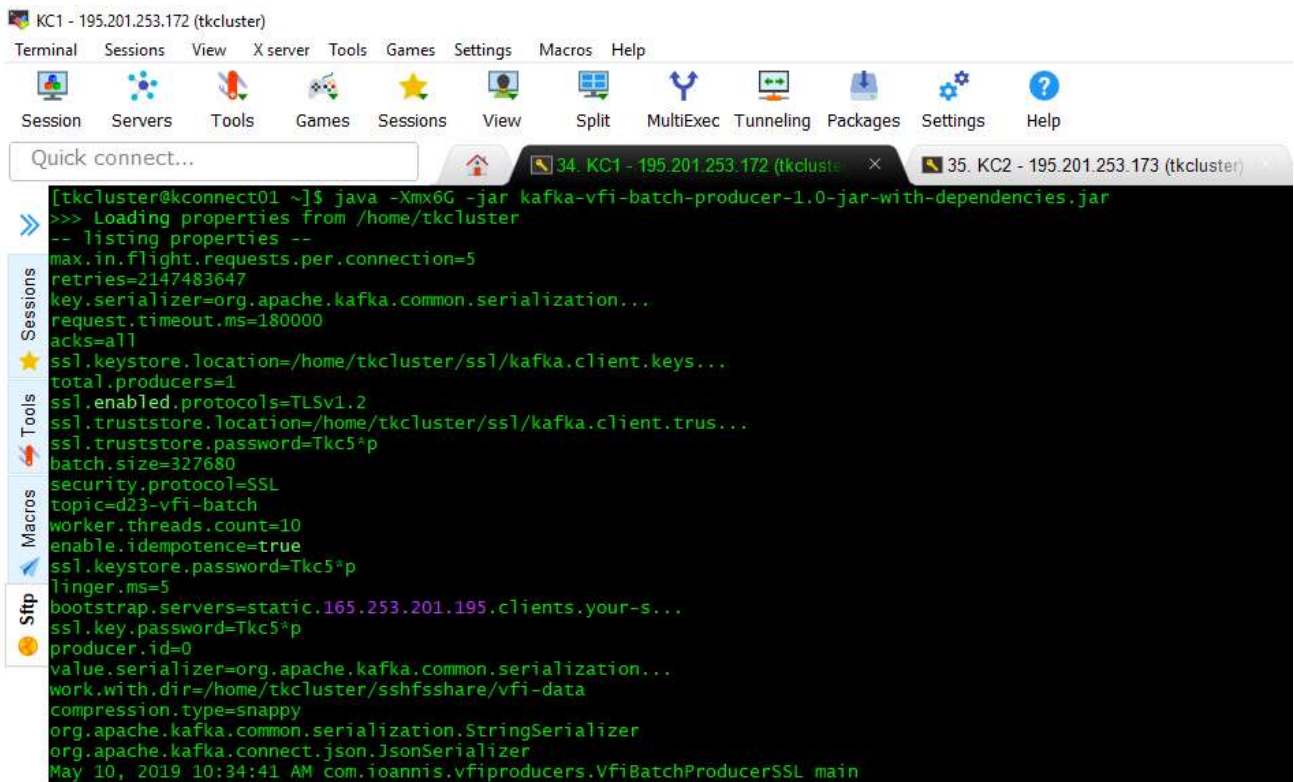
2. At this point the `vfibatchproducer.properties` file, where the jar loads properties from, should be edited. The topic just created should be set accordingly (in this example “d23-vfi-batch”). Also, the directory where the data to be loaded reside should be valid.
3. The Connector can be started by using the command below. The memory it is allowed to use can be increased for better performance.

```
[wp3user04@wp3node01 ~]$ java -jar kafka-vfi-batch-producer-1.0-jar-with-dependencies.jar_
```

Figure 5.10 VFI batch connector startup command

Below a single connector can be seen while starting up:

D2.3 Development of Toolboxes Integration Connectors



The screenshot shows a terminal window titled "KC1 - 195.201.253.172 (tkcluster)". The terminal displays the output of a Java command: `java -Xmx6G -jar kafka-vfi-batch-producer-1.0-jar-with-dependencies.jar`. The output shows the connector loading properties from `/home/tkcluster` and listing various configuration properties. The properties include `max.in.flight.requests.per.connection=5`, `retries=2147483647`, `key.serializer=org.apache.kafka.common.serialization...`, `request.timeout.ms=180000`, `acks=all`, `ssl.keystore.location=/home/tkcluster/ssl/kafka.client.keys...`, `total.producers=1`, `ssl.enabled.protocols=TLSv1.2`, `ssl.truststore.location=/home/tkcluster/ssl/kafka.client.trus...`, `ssl.truststore.password=Tkc5*p`, `batch.size=327680`, `security.protocol=SSL`, `topic=d23-vfi-batch`, `worker.threads.count=10`, `enable.idempotence=true`, `ssl.keystore.password=Tkc5*p`, `linger.ms=5`, `bootstrap.servers=static.165.253.201.195.clients.your-s...`, `ssl.key.password=Tkc5*p`, `producer.id=0`, `value.serializer=org.apache.kafka.common.serialization...`, `work.with.dir=/home/tkcluster/sshfsshare/vfi-data`, `compression.type=snappy`, `org.apache.kafka.common.serialization.StringSerializer`, `org.apache.kafka.connect.json.JsonSerializer`, and the timestamp `May 10, 2019 10:34:41 AM com.ioannis.vfi-producers.VfiBatchProducerSSL main`.

Figure 5.11 VFI batch connector startup output

4. Incoming message rates for the topic created can be observed in the Grafana Cluster Overview. The image below shows interesting information about the performance of the connector just started:

D2.3 Development of Toolboxes Integration Connectors

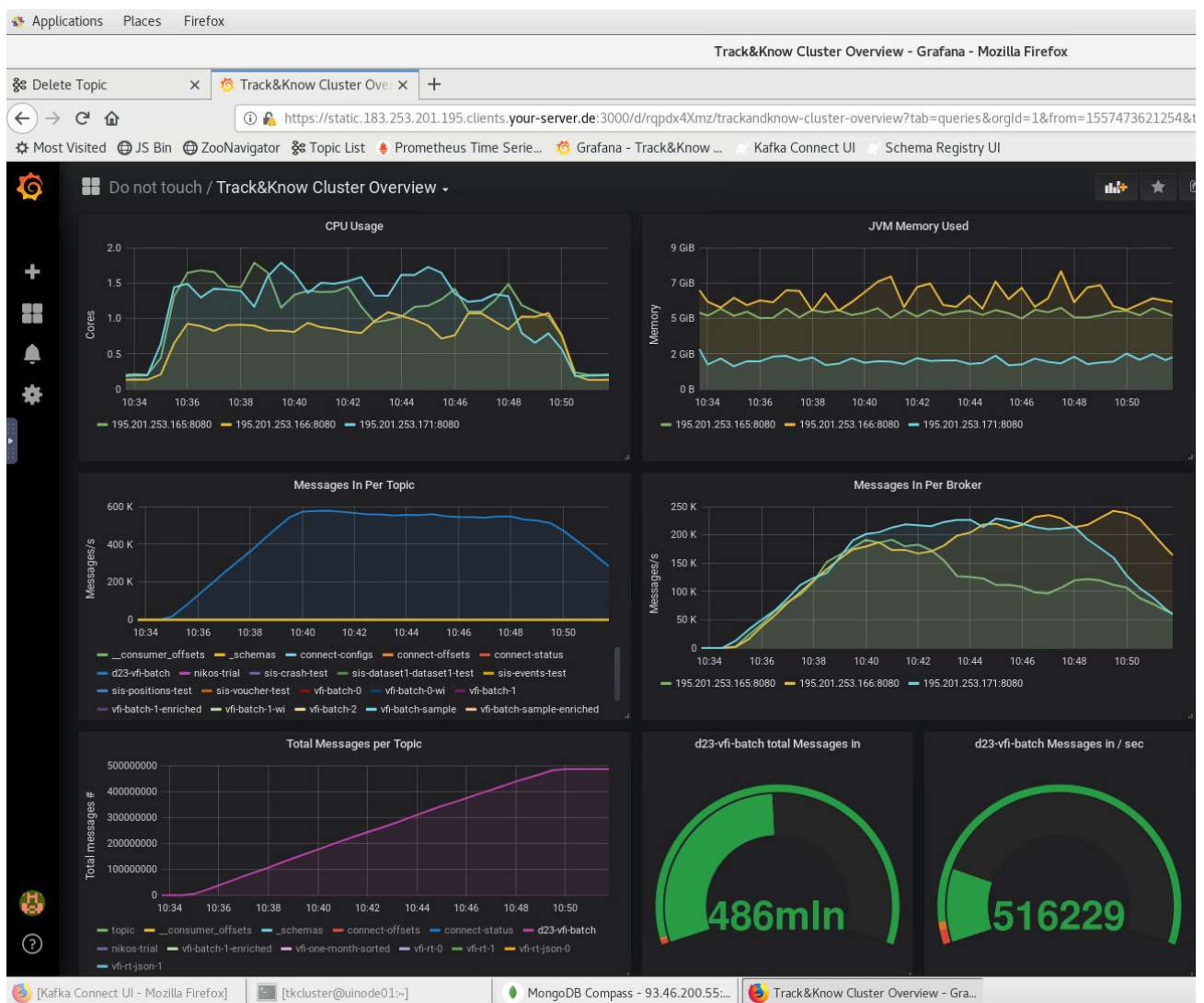


Figure 5.12 Incoming message rates for VFI batch data

The CPU usage shown in the top left graph indicates the cores utilised on the 3 Apache Kafka Brokers that exist in the Track & Know Platform. Out of the total 8 vCPUs per node (total of 24 Cores) only roughly 4 cores are used (something that we have noticed to stay at this level even with multiple data loading tasks). On the right it can be seen that the “JVM Memory Used” is also very low. The Brokers have 32GB of RAM available each (total of 96GB), with only about 14GB used in this run. It can be seen in the graph titled “Messages in Per Topic” above that the message rates climb up to 500 – 550 thousand messages per second. In the middle right the graph with title “Messages in Per Broker” shows that the messages are handled by all three brokers. The lower left graph with title “Total Messages per Topic” is focused on the “d23-vfi-batch” topic which is used for this deliverable test and it shows the messages increasing in the topic to reach a total of 486 million messages before the experiment is killed.

In this experiment it should be noted that the message rates achieved are with encryption and compression enabled and also the replication factor is set to 2 which means that the messages are copied twice between brokers for high availability.

D2.3 Development of Toolboxes Integration Connectors

Since the Producer Connector for the VFI batch data can be run in parallel it is interesting to present a run where multiple instances are started on multiple hosts. This can be achieved by starting the connectors as shown below where 5 instances are started:

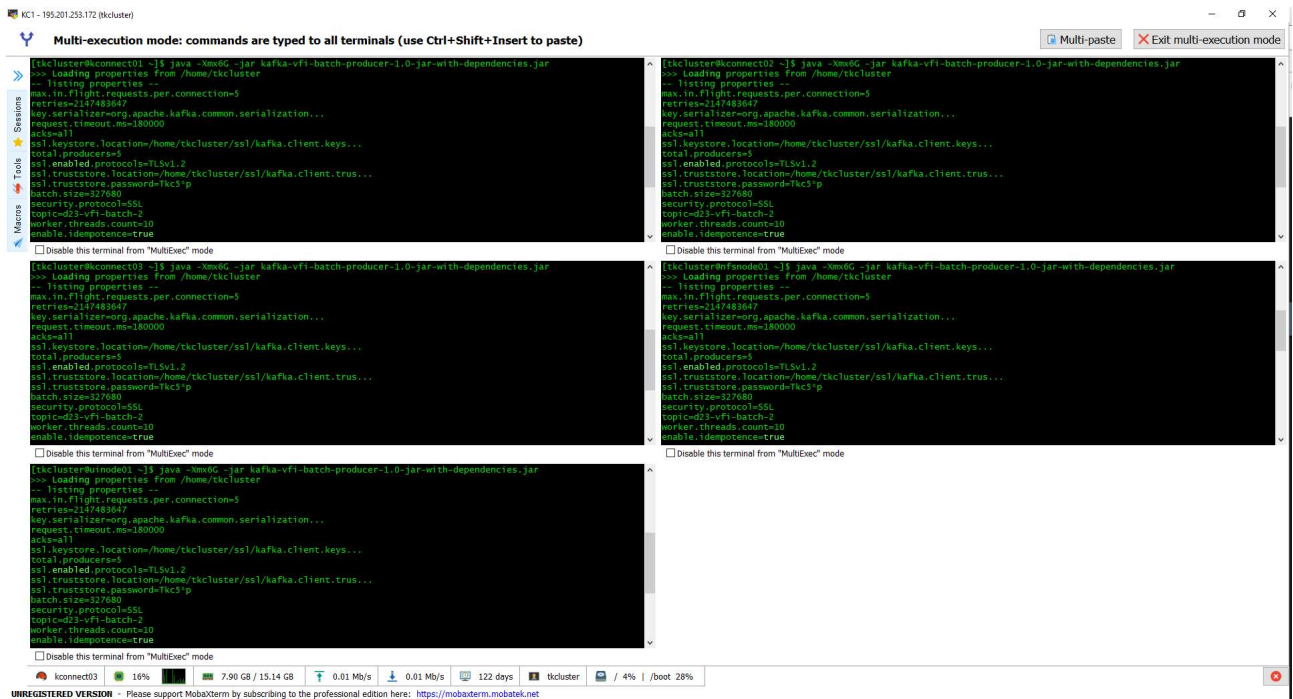


Figure 5.13 Multiple producers startup

Incoming message rates for the topic created can be observed in the Grafana Cluster Overview. The image below shows interesting information about the performance of the connectors just started:

D2.3 Development of Toolboxes Integration Connectors

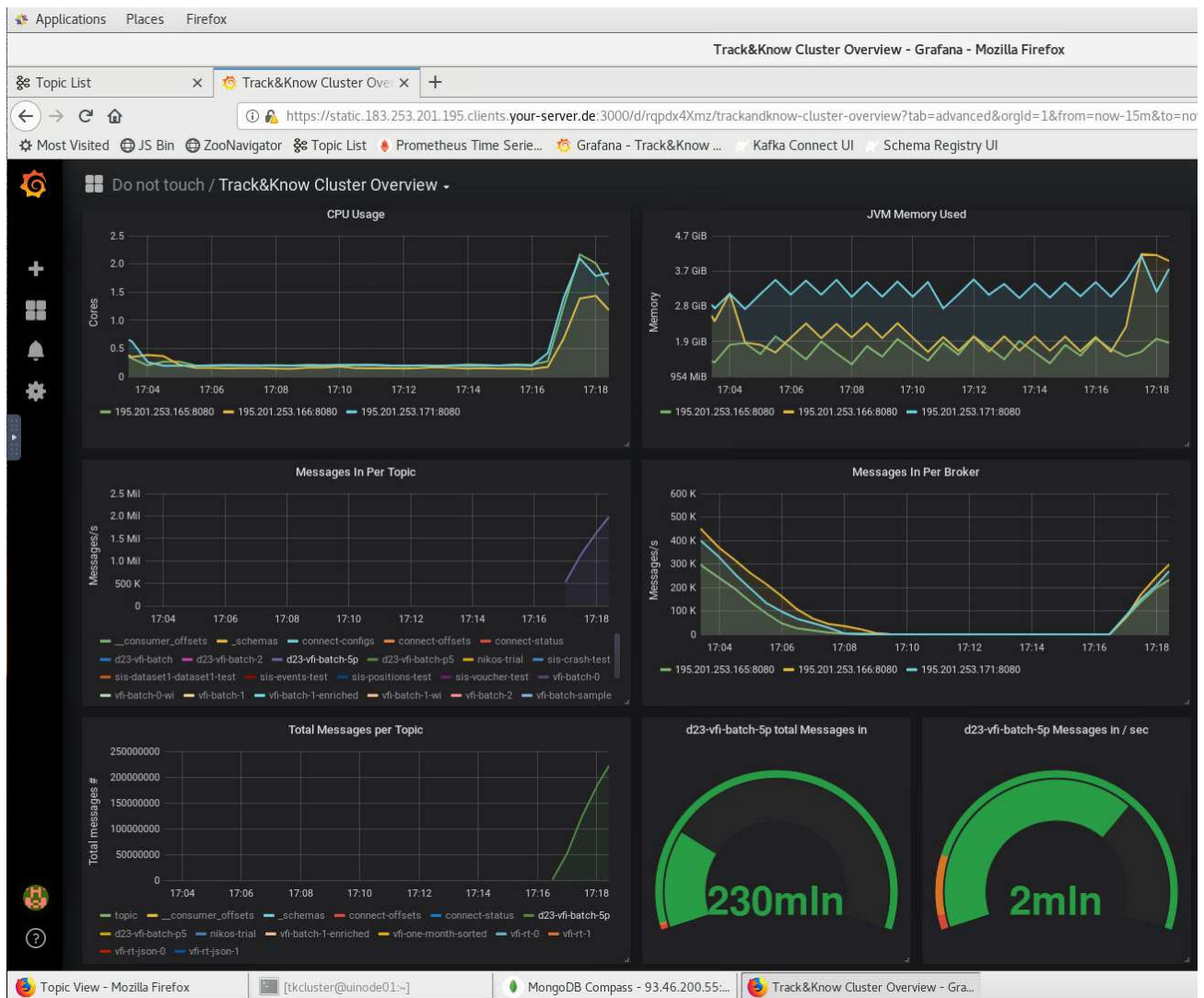


Figure 5.14 Incoming message rates with multiple producers

The CPU usage shown in the top left graph indicates that it remained at the same levels even though the message rate has more than tripled. The same thing holds for the “JVM Memory Used” which has no significant change. The message rate reaches a peak of **2 Million messages per second** where it begins to drop as the multiple threads finish their portion of the task and release resources. In cases where more data are to be processed, the message rate is sustained and will rise further. **It is noteworthy that the total of about 542 Million messages contained in the historical dataset provided by VFI (about 100GB of data) is loaded in under 8 minutes. This indicates that billions of messages can be processed within an hour. It should be noted that the message number is more important to the message size as it is more challenging to handle many small messages vs fewer larger ones and that the message rates would have been significantly higher if the encryption that the Track&Know use cases dictate was not needed.**

The figure below shows the graphs when the same task shown above has finished:

D2.3 Development of Toolboxes Integration Connectors

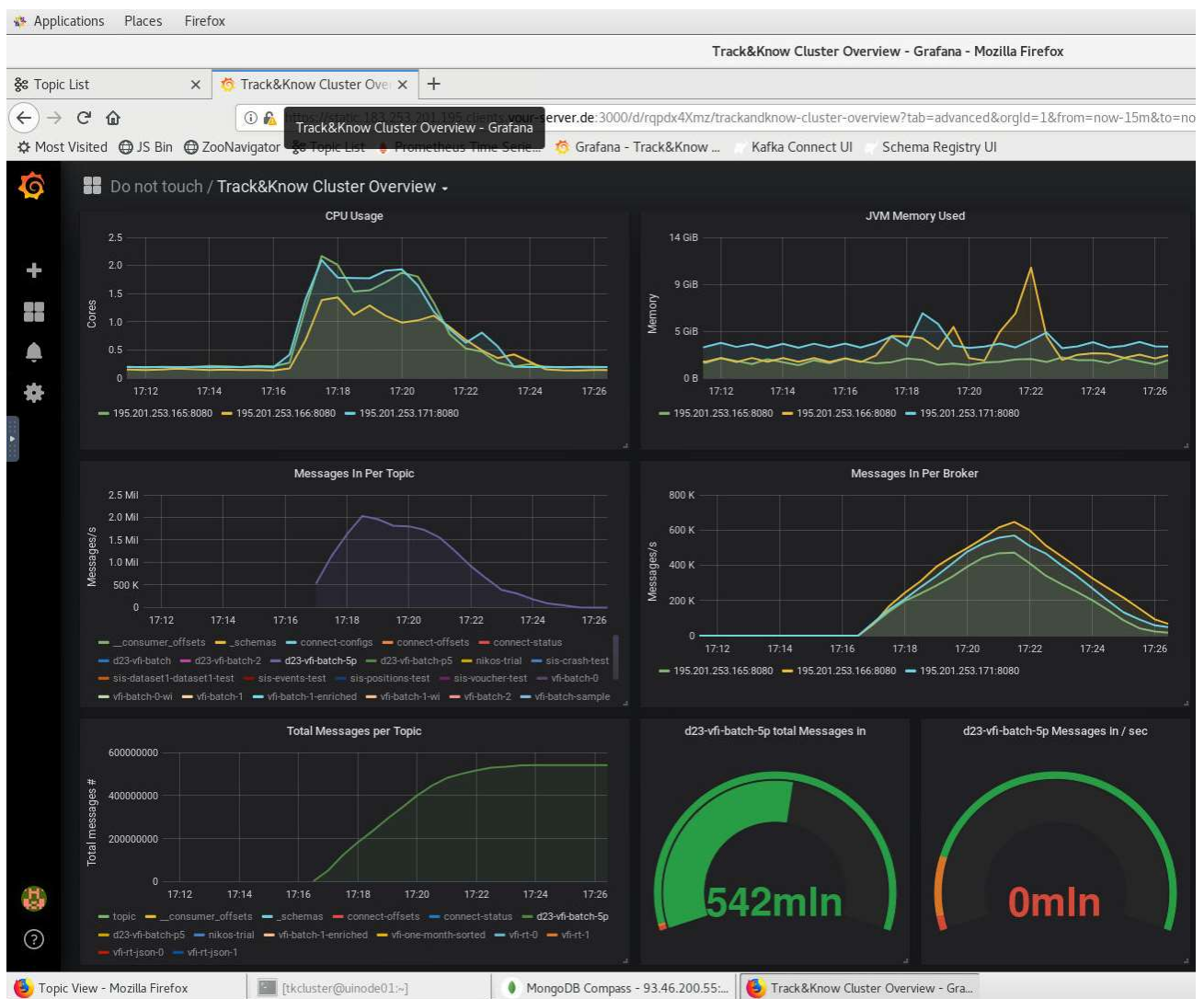


Figure 5.15 Multiple producers (experiment finished)

A comparison between a single and multiple connector runs can be seen below.

Table 4 VFI batch data loading runs

Run 1	Producers	Threads per Producer	Message Rate
	1	10	516 Thousand messages per second
Run 2	Producers	Threads per Producer	Message Rate
	5	10	2 Million messages per second

During the loading or once the data is loaded in the topic of choice, it can be read by clients for subsequent processing. By using the command below, JSON messages can be retrieved (read) from the topic of choice (here vfi-batch-1 – similar for vfi-batch-0) and displayed in the console:



```
[wp3user05@wp3node01 ~]$ ~/confluent-5.0.0/bin/kafka-console-consumer --bootstrap-server static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093 --topic vfi-batch-1 --from-beginning --consumer.config /home/wp3user05/connectors/client.properties_
```

Figure 5.16 Reading VFI batch data from topic

Sample of String (CSV) data as they exist in topic vfi-batch-0 and topic vfi-batch-1:

```
37200;12400_103480;2018-07-01 00:06:01.000;3;;33.050337;34.686815;0;0;0;0;357;;CANBUS;100;;;0;FM1120;Passenger Car;;
```

Figure 5.17 Sample of VFI batch data message

Please note that the empty values in the CSV represent fields that were not available/not set. The column names applicable to that data are the same that apply to the original CSV files provided by VFI:

```
company;vehicle;localDate;engineStatus;driver;driverEvent;longitude;latitude;altitude;angle;speed;odometer;satellites;fuelLevelLt;countryCode;rpm;levelType;fuelTankSize;vehicleOdometer;fuelConsumed;engineHours;closeToGasStation;deviceType;VehicleType;fuelRawValue;
```

5.1.2 Kafka Connect type Connector for the VFI live data

For the purpose of introducing the VFI live data to the Track&Know Platform the Kafka Connect functionality was used. More specifically a custom Kafka Connect Source Connector was developed which is deployed in a highly available Connect Cluster. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the VFI live data and load them into the topic of choice according to the code of the connector and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the VFI live data will continue to enter the Platform. At configurable intervals the connector pulls the live data from a provided REST API and for a set number of vehicles. The data retrieved are then written to a Kafka topic of choice. The VFI API provides the data in JSON format and the connector supports serialisation in both JSON and Avro. Authentication and encryption settings are already setup between the Kafka Connect Workers and the Kafka Brokers and no setup is necessary for individual connectors.

The connector and other required jars are built using `mvn clean compile package` which produces a target folder containing the jars. This folder is then moved to the dedicated directory of each Worker node of the Kafka Connect Cluster and this way the connector code is available and can be instantiated.

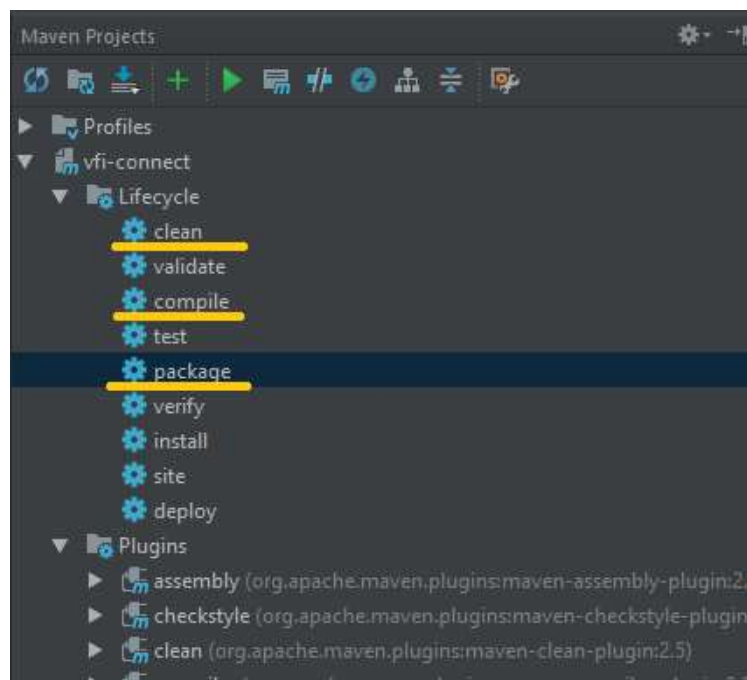


Figure 5.18 Building the VFI live data connector

The contents of the connectors directory (here connect-jars) can be seen below. The VFI live data connector exists in the vfi-connect-test-1.0.0 directory:

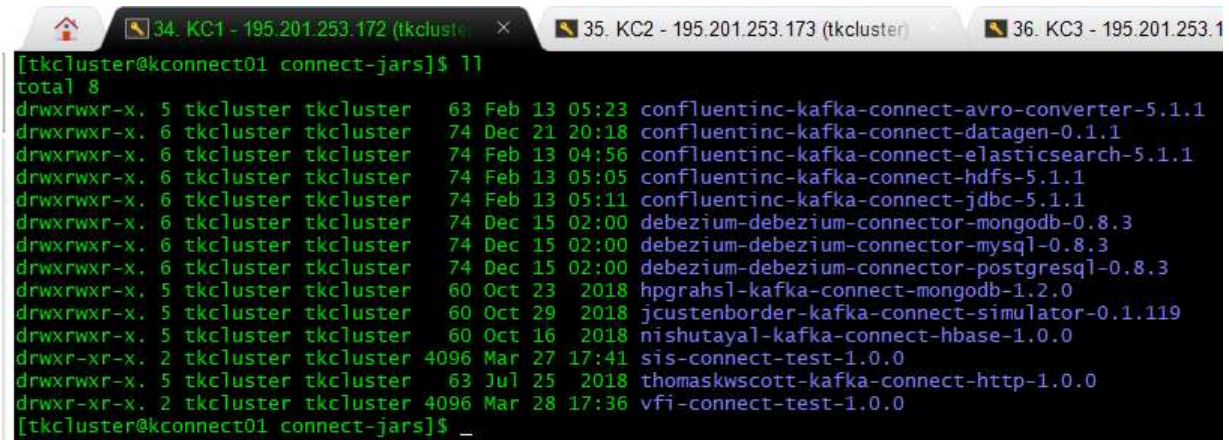


Figure 5.19 Kafka Connect jars directory

For the purpose of starting a Kafka Connect type of connector, the Kafka Connect UI component is set up where all the connectors inside the Workers directory and the running connectors can be seen:

D2.3 Development of Toolboxes Integration Connectors

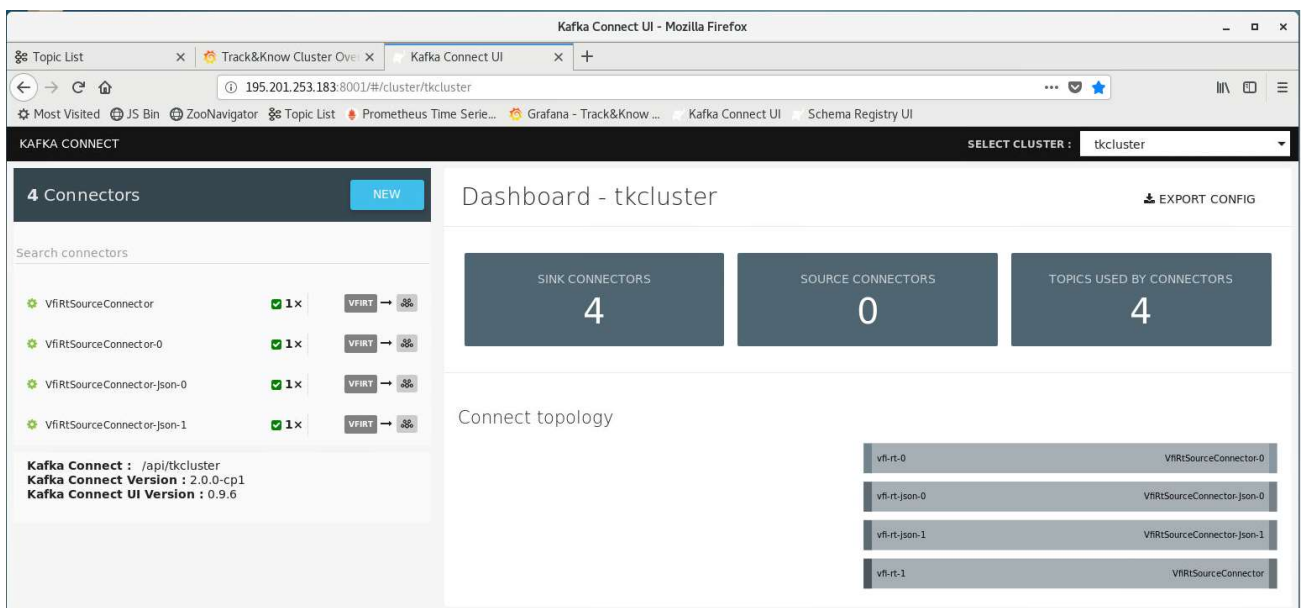


Figure 5.20 Kafka Connect UI for starting connectors

The available installed connectors can be accessed by pressing the “NEW” button. After Selecting a particular connector (Here the VFI connector) the configuration has to be entered.

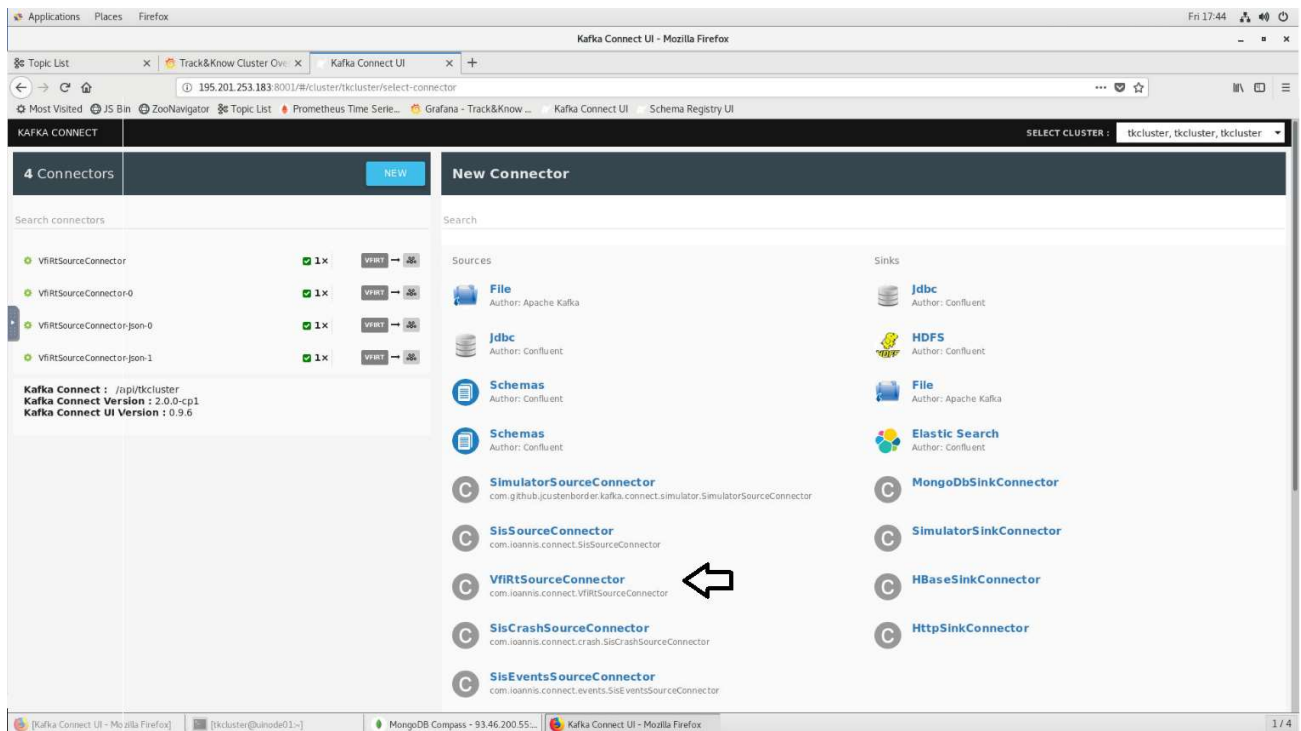


Figure 5.21 Selecting a new Connector

In the figure below the VfiRtSourceConnector has been selected and its configuration window is shown:

D2.3 Development of Toolboxes Integration Connectors

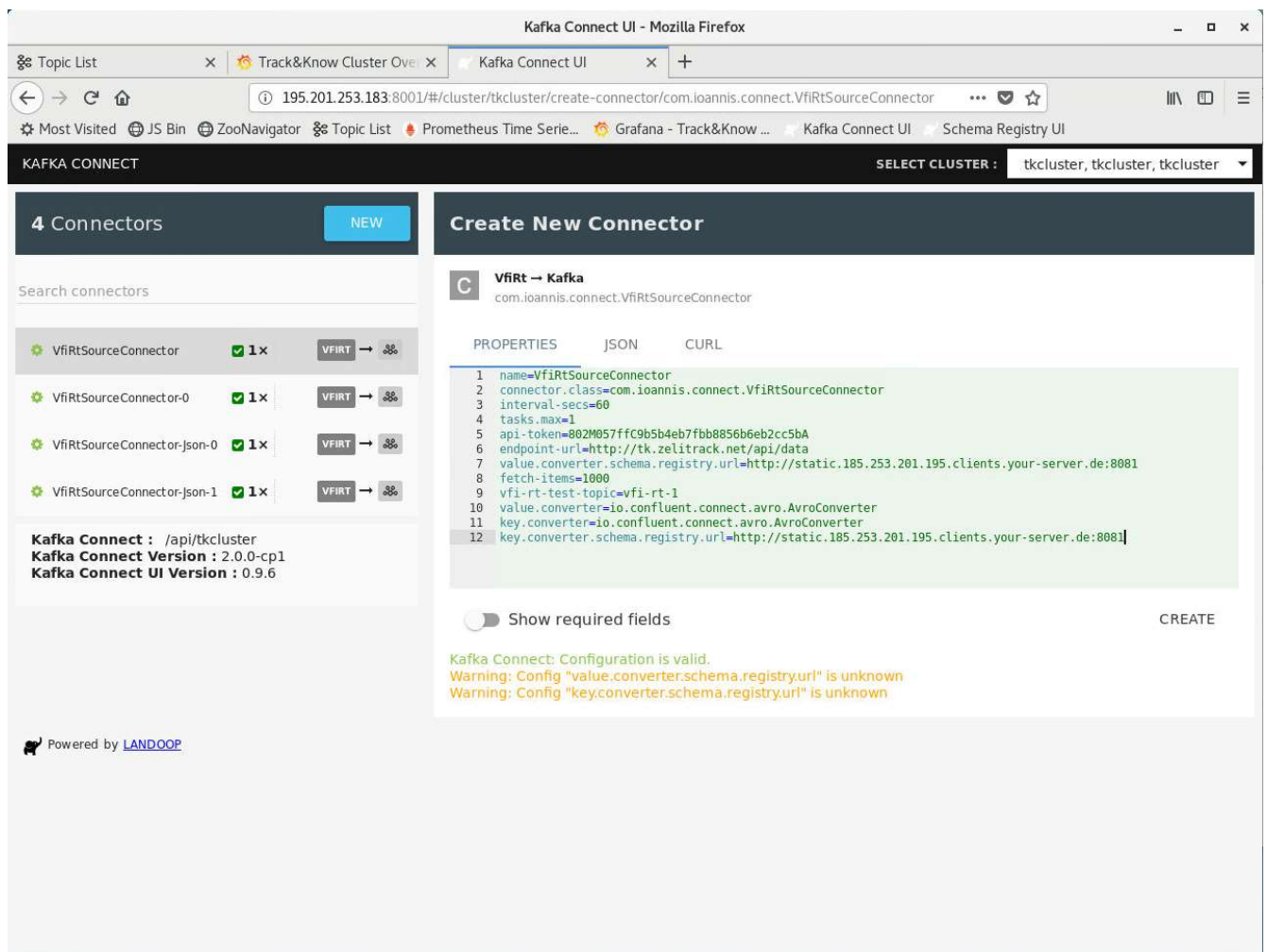


Figure 5.22 Configuring a Kafka Connect Connector

In detail, the configuration options are listed in the following table together with their description:

Table 5 Configuration options for the VFI live data connector

Property and Sample value (Example)	Description
name=VfiRtSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.VfiRtSourceConnector	The type of connector which is actually the connector's java class
interval-secs=60	How often data should be retrieved from the VFI API. At this stage the minimum should be 60 seconds.
tasks.max=1	The number of tasks this connector should start.
api-token=sampleToken7fbb8856b6eb	The API token used – Provided by VFI.

endpoint-url=http://tk.zelitrack.net/api/data	The URL of the REST endpoint that provides the data.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The schema registry url where the value schema is held.
fetch-items=1000	The number of items (vehicles) for which data will be returned. The maximum allowed is 1000
vfi-rt-topic=vfi-rt-1	The topic where the retrieved data will be written in Kafka
value.converter=io.confluent.connect.avro.AvroConverter	The Converter used for the value. Here is Avro but can also be JSON
key.converter=io.confluent.connect.avro.AvroConverter	The Converter used for the Key. Here is Avro but can also be JSON
key.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The schema registry url where the key schema is held.

Currently there are 4 Kafka Connect VFI Connectors which have been running continuously for a time period of about 80 days. These connectors are accumulating vehicle data in JSON and Avro formats with a single and multiple partitions in the target topics:

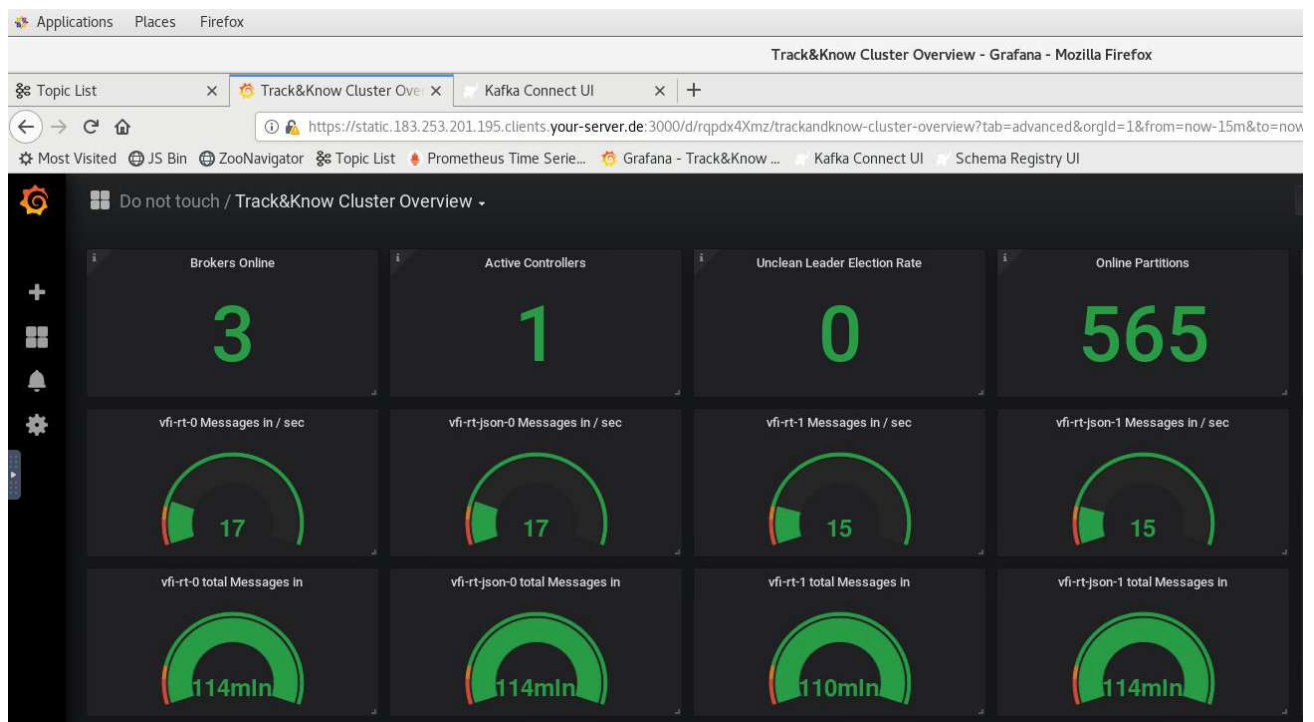
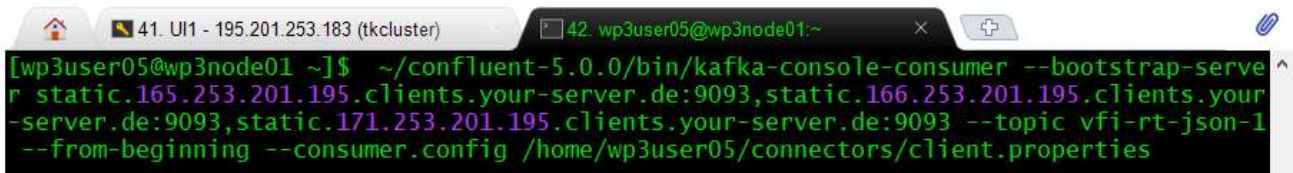


Figure 5.23 VFI live data connectors running

As it can be seen in the figure above, the total number of messages is above 110 million. Toolboxes can access that data from their beginning or any other offset and at whatever rate is applicable. Of course, a client can connect to the above live topics and begin receiving new data from that point onwards.

By using the command below, JSON messages can be retrieved (read) from the topic of choice (here vfi-rt-json-1 – similar for vfi-rt-json-0) and displayed in the console:



```
[wp3user05@wp3node01 ~]$ ~/confluent-5.0.0/bin/kafka-console-consumer --bootstrap-server static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093 --topic vfi-rt-json-1 --from-beginning --consumer.config /home/wp3user05/connectors/client.properties
```

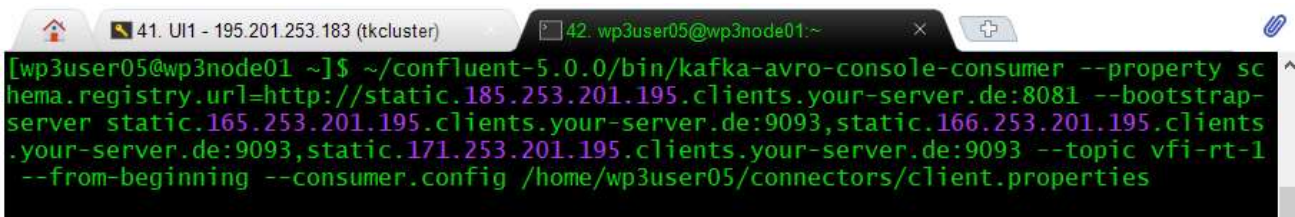
Figure 5.24 Reading VFI live data messages

Sample of JSON data as they exist in topics vfi-rt-json-0 and topics vfi-rt-json-1:

```
{
  "company": "27150",
  "vehicle": "9050_180060",
  "localDate": "2019-02-15 16:48:47.000",
  "engineStatus": "1",
  "driver": "",
  "driverEvent": "",
  "longitude": "19.472450",
  "latitude": "41.318013",
  "altitude": "0",
  "angle": "50",
  "speed": "31",
  "odometer": 19,
  "satellites": 17,
  "fuelLevelLt": "",
  "countryCode": "355",
  "rpm": "",
  "levelType": "Floater",
  "fuelTankSize": "null",
  "vehicleOdometer": "",
  "fuelConsumed": "",
  "engineHours": "",
  "closeToGasStation": "0",
  "deviceType": "FMB120",
  "VehicleType": "Truck 3,5 t",
  "fuelRawValue": "",
  "OdometerNotNull": 19,
  "odometerStr": "19",
  "satellitesNotNull": 17,
  "satellitesStr": "17"
}
```

Figure 5.25 Sample of VFI live JSON data

By using the command below AVRO messages can be retrieved (read) from the topic of choice (here vfi-rt-1 – similar for vfi-rt-json-0) and displayed in the console:



```
[wp3user05@wp3node01 ~]$ ~/confluent-5.0.0/bin/kafka-avro-console-consumer --property schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081 --bootstrap-server static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093 --topic vfi-rt-1 --from-beginning --consumer.config /home/wp3user05/connectors/client.properties
```

Figure 5.26 Reading VFI live data messages (AVRO)

Sample of AVRO data as they exist in topics vfi-rt-0 and topics vfi-rt-1:

```
{
  "company": "50250",
  "vehicle": "16750_144860",
  "localDate": "2019-02-15 17:48:26.000",
  "engineStatus": "1",
  "driver": "",
  "driverEvent": "",
  "longitude": "23.066393",
  "latitude": "40.750315",
  "altitude": "104",
  "angle": "223",
  "speed": "11",
  "odometer": 8,
  "satellites": 16,
  "fuelLevelLt": "",
  "countryCode": "30",
  "rpm": "",
  "levelType": "Floater",
  "fuelTankSize": "0",
  "vehicleOdometer": "",
  "fuelConsumed": "",
  "engineHours": "",
  "closeToGasStation": "0",
  "deviceType": "FMA120",
  "VehicleType": "Bus",
  "fuelRawValue": "",
  "OdometerNotNull": 8,
  "odometerStr": "8",
  "satellitesNotNull": 16,
  "satellitesStr": "16"
}
```

Figure 5.27 Sample of VFI live AVRO data

Empty fields represent values that did not exist.

Please note that while the data appear to be exactly the same as the JSON in the topics shown above, the main difference with AVRO is that the Schema and Keys of the values within messages are not transmitted by the producing and/or consuming end with each message as they are retrieved from the Schema Registry, reducing thus overhead:

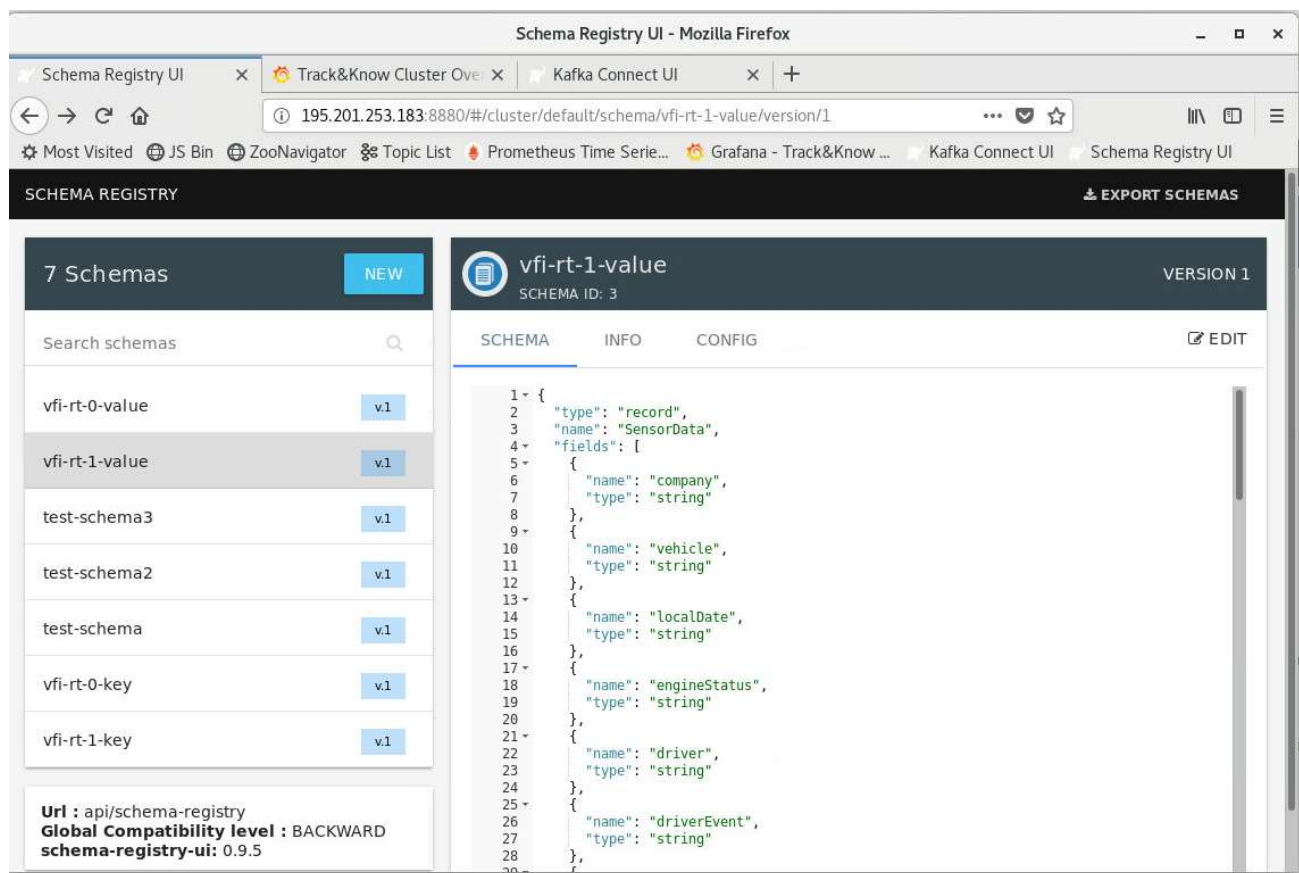


Figure 5.28 Schema Registry entry for VFI live data

This section concludes the description of the connectors developed for the purpose of introducing the VFI provided datasets and live data into the Track&Know Platform.

5.2 SIS Data Connectors

In this section the connectors implemented for introducing the SIS data to the Track&Know platform are discussed, presenting information about their usage, configuration, startup and monitoring. The connectors developed for this purpose are based on Kafka Connect technology and therefore in order to introduce the SIS historical data to the Track&Know Platform the Kafka Connect functionality was used. More specifically, several custom Kafka Connect Source Connectors were developed which deal with the data provided, with the latter organised in a MongoDB standalone instance on SIS premises. The connectors are deployed in the highly available Connect Cluster in the Track&Know Platform. Once a connector is instantiated the Connect Cluster Workers perform the task of connecting to the SIS MongoDB instance, enable compression and batching and proceeds to retrieve the data (for which the connector in question was developed), finally loading them into the topic of choice according to the connector configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the SIS data will continue to enter the Platform as the loading is resumed by another Worker.

The connector implementation for all SIS data connectors that will be presented in the following subsections supports multiple tasks per connector. This means that the connectors are capable of starting automatically and according to their configuration, more than one identical task, assigned to the Kafka Connect cluster Workers. The implemented logic within the connectors indicates the way that the loading task should be split among the multiple tasks of a connector. This effectively splits and parallelises the loading of data into a topic according to

the number of tasks. While the maximum number of tasks is in theory infinite, it should be made clear that as the connectors and tasks increase the underlying number of available workers must be able to support them. This can of course be achieved by introducing more Kafka Connect Worker nodes in the Track&Know Platform which allows the processing of tasks to be re-balanced across the Workers and therefore allows horizontal scalability to be achieved.

The data retrieved are written to Kafka topics of choice and according to the connector configuration. Especially for the cases of the SIS datasets the Kafka topics used are set to have a small retention period which guarantees that the datasets provided and loaded to the Track&Know Platform will never be stored permanently in its entirety according to what the signed agreements between SIS and other partners dictate. The SIS data exist as MongoDB documents and are stored in the topics in JSON format. Authentication and encryption settings are already setup between the Kafka Connect Workers and the Kafka Brokers and no setup is necessary for individual connectors. The connectors are also implemented with exactly once delivery semantics in mind. This ensures that data read from the relative MongoDB collections will not result in duplicate messages within the target Kafka Topic. This is achieved by maintaining a source index which is stored together with each message in Kafka and allows processing to be idempotent and be resumed if necessary.

Similar to the other Kafka Connect code presented, the connectors and other required jars are built using mvn clean compile package which produces a target folder containing the jars. This folder is then moved to the dedicated directory of each Worker node of the Kafka Connect Cluster and this way the connector code is available and can be instantiated.

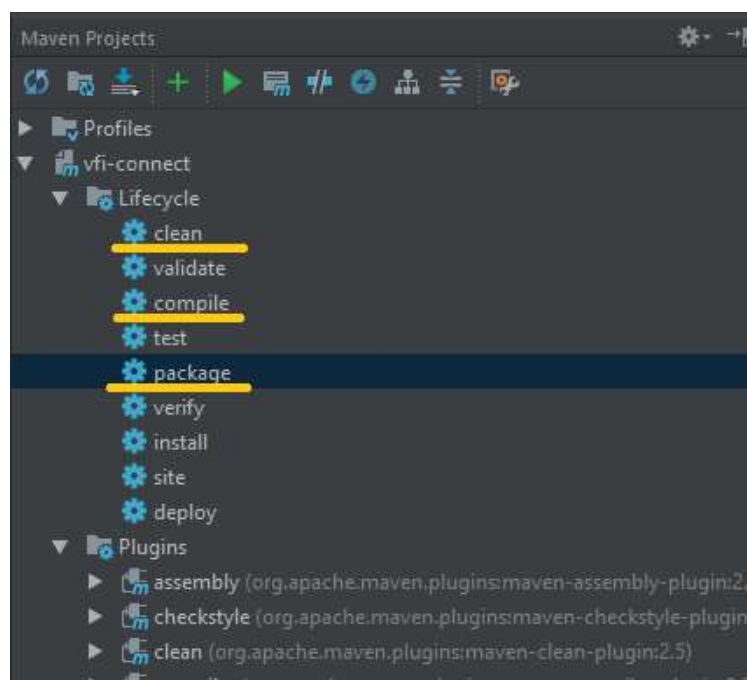


Figure 5.29 Building SIS connectors

The contents of the connectors directory (here connect-jars) can be seen below. The SIS connectors exist in the sis-connect-test-1.0.0 directory:

```

[tkcluster@kconnect01 connect-jars]$ ll
total 8
drwxrwxr-x. 5 tkcluster tkcluster 63 Feb 13 05:23 confluentinc-kafka-connect-avro-converter-5.1.1
drwxrwxr-x. 6 tkcluster tkcluster 74 Dec 21 20:18 confluentinc-kafka-connect-datagen-0.1.1
drwxrwxr-x. 6 tkcluster tkcluster 74 Feb 13 04:56 confluentinc-kafka-connect-elasticsearch-5.1.1
drwxrwxr-x. 6 tkcluster tkcluster 74 Feb 13 05:05 confluentinc-kafka-connect-hdfs-5.1.1
drwxrwxr-x. 6 tkcluster tkcluster 74 Feb 13 05:11 confluentinc-kafka-connect-jdbc-5.1.1
drwxrwxr-x. 6 tkcluster tkcluster 74 Dec 15 02:00 debezium-debezium-connector-mongodb-0.8.3
drwxrwxr-x. 6 tkcluster tkcluster 74 Dec 15 02:00 debezium-debezium-connector-mysql-0.8.3
drwxrwxr-x. 6 tkcluster tkcluster 74 Dec 15 02:00 debezium-debezium-connector-postgresql-0.8.3
drwxrwxr-x. 5 tkcluster tkcluster 60 Oct 23 2018 hpgrahsl-kafka-connect-mongodb-1.2.0
drwxrwxr-x. 5 tkcluster tkcluster 60 Oct 29 2018 jcusterborder-kafka-connect-simulator-0.1.119
drwxrwxr-x. 5 tkcluster tkcluster 60 Oct 16 2018 nishutayal-kafka-connect-hbase-1.0.0
drwxr-xr-x. 2 tkcluster tkcluster 4096 Mar 27 17:41 sis-connect-test-1.0.0
drwxrwxr-x. 5 tkcluster tkcluster 63 Jul 25 2018 thomaskwscott-kafka-connect-http-1.0.0
drwxr-xr-x. 2 tkcluster tkcluster 4096 Mar 28 17:36 vfi-connect-test-1.0.0
[tkcluster@kconnect01 connect-jars]$

```

Figure 5.30 Kafka Connect jars directory with SIS connectors

5.2.1 Kafka Connect type Connector for the SIS DATASET1 data

The data provided by SIS which contain the main mobility information with the respective vehicle GPS locations were made available in a MongoDB instance which was setup in their (SIS) premises and it is accessed over the network with the credentials provided. The database and related collection are named dataset1 and DATASET1 respectively. A small portion of the data can be seen in the screenshot below taken from the MongoDB Compass Client while the latter is connected to the remote database and collection:

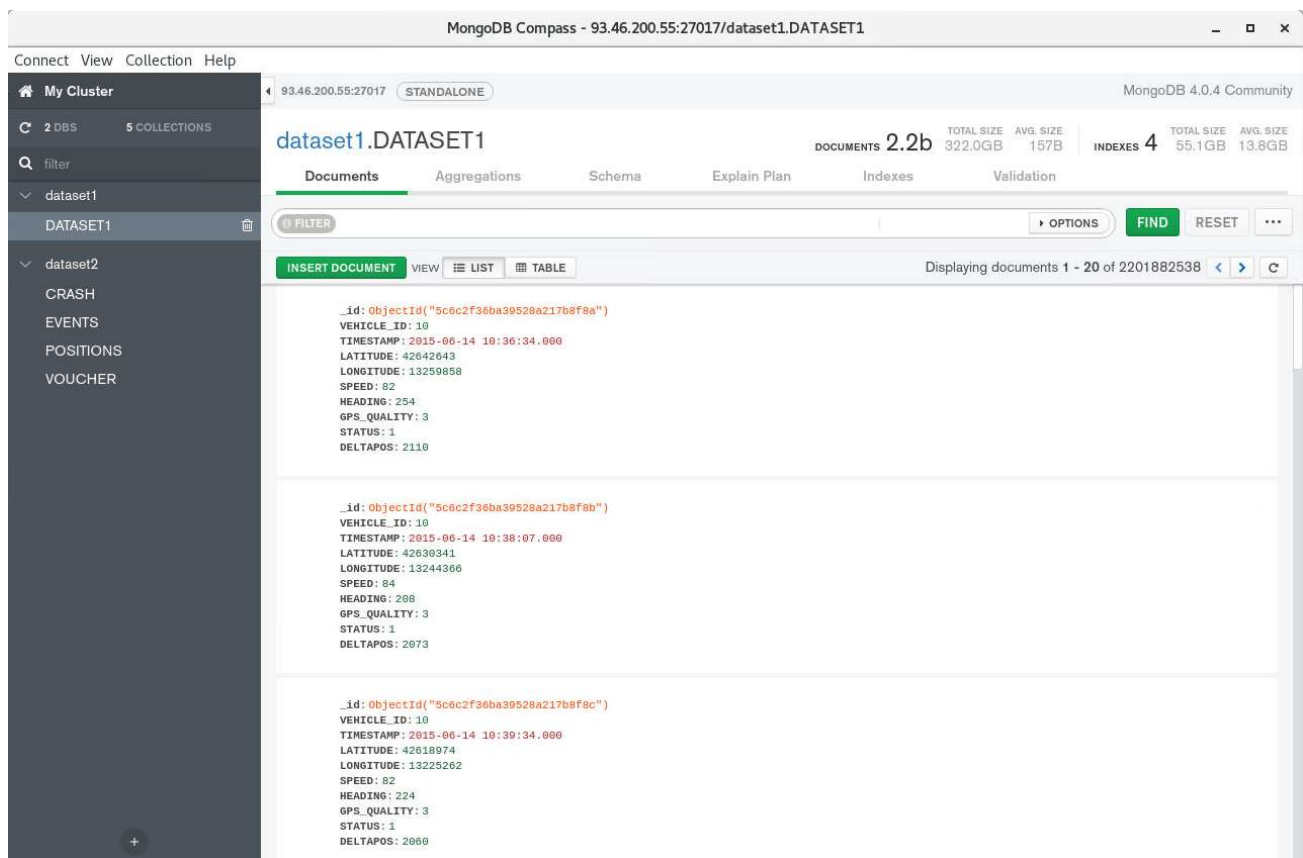


Figure 5.31 SIS DATASET1 collection

D2.3 Development of Toolboxes Integration Connectors

For the purpose of starting a SIS Kafka Connect type of connector, the Kafka Connect UI component is used all the connectors inside the Workers directory and the running connectors can be seen:

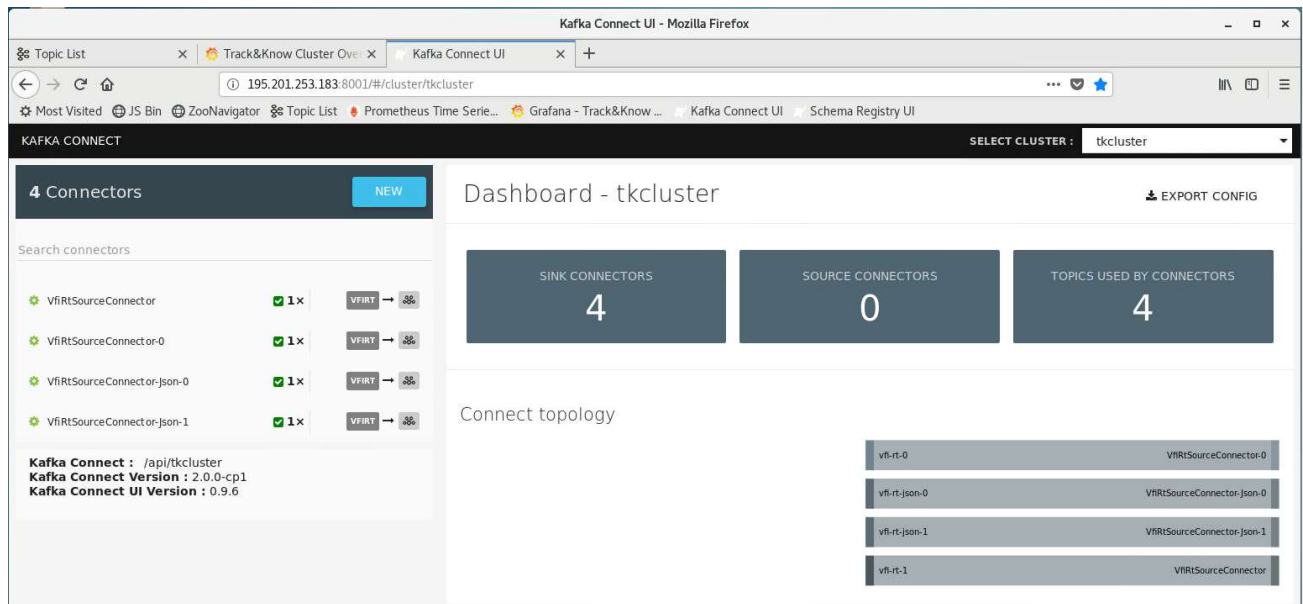


Figure 5.32 Kafka Connect UI and running connectors

The available installed connectors can be accessed by pressing the “NEW” button. After Selecting a particular connector (Here the SisSource connector) the configuration has to be entered.

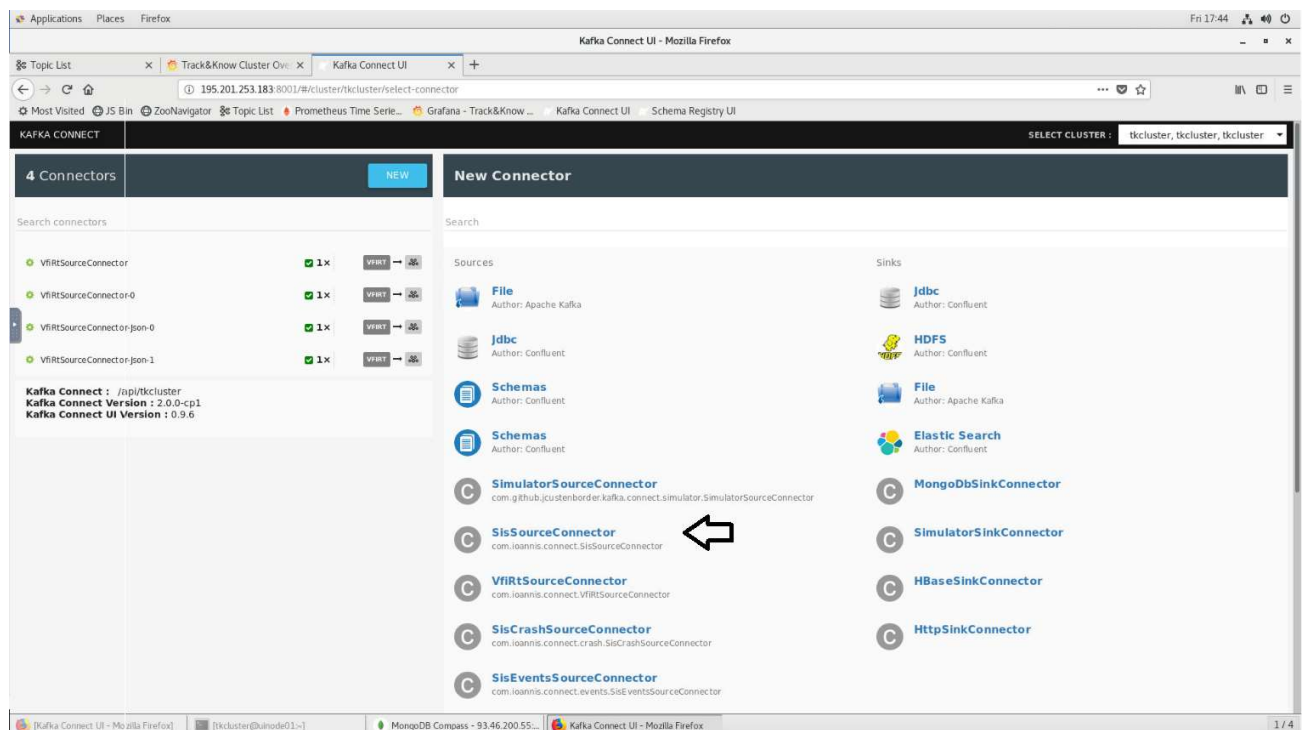


Figure 5.33 Selecting the DATASET1 connector

In the figure below the SisSourceConnector has been selected and its configuration window is shown:

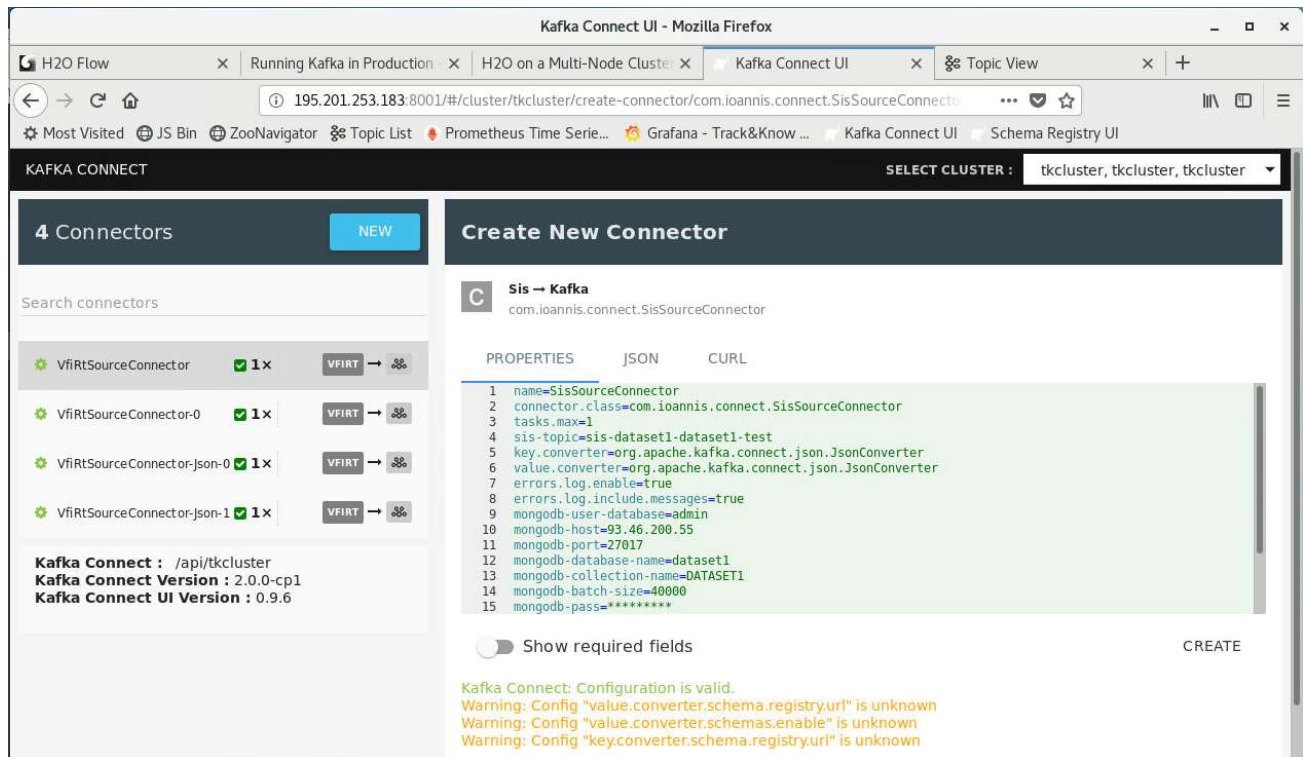


Figure 5.34 Configuring the DATASET1 connector

In detail, the configuration options are listed in the following table together with their description:

Table 6 SIS DATASET1 connector configuration options

Property and Sample value (Example)	Description
name=SisSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.SisSourceConnector	The type of connector which is actually the connector's java class
tasks.max=1	The max number of tasks this connector should start.
sis-topic=d23-sis-dataset1	The topic where the retrieved data will be written in Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message key (here JSON is used)
value.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message value (here JSON is used)

errors.log.enable=true	Enables or disables error logging
errors.log.include.messages=true	Controls whether the error messages are included in the logs
mongodb-user-database=admin	The database collection to authenticate against
mongodb-host=93.46.200.55	The host IP where the MongoDB instance is running
mongodb-port=27017	The host port where the MongoDB instance is running
mongodb-database-name=dataset1	The database name where the collection is in
mongodb-collection-name=DATASET1	The name of the collection to be read
mongodb-batch-size=40000	The batch size that will be used
mongodb-pass=*****	The password to access the database
kafka-send-every-num-of-records=40000	The number of records that when gathered will be sent to Kafka.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the value schema registry
value.converter.schemas.enable=false	Setting to control whether the schema should also be sent
key.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the key schema registry

5.2.2 Kafka Connect type Connector for the SIS CRASH data

The data provided by SIS which contain information about accidents which have occurred and are related to the dataset were also made available in the same MongoDB instance setup in SIS premises and accessed over the network with provided credentials. The database and related collection are named dataset2 and CRASH respectively. A small portion of the data can be seen in the screenshot below taken from the MongoDB Compass Client while the latter is connected to the remote database and collection:

D2.3 Development of Toolboxes Integration Connectors

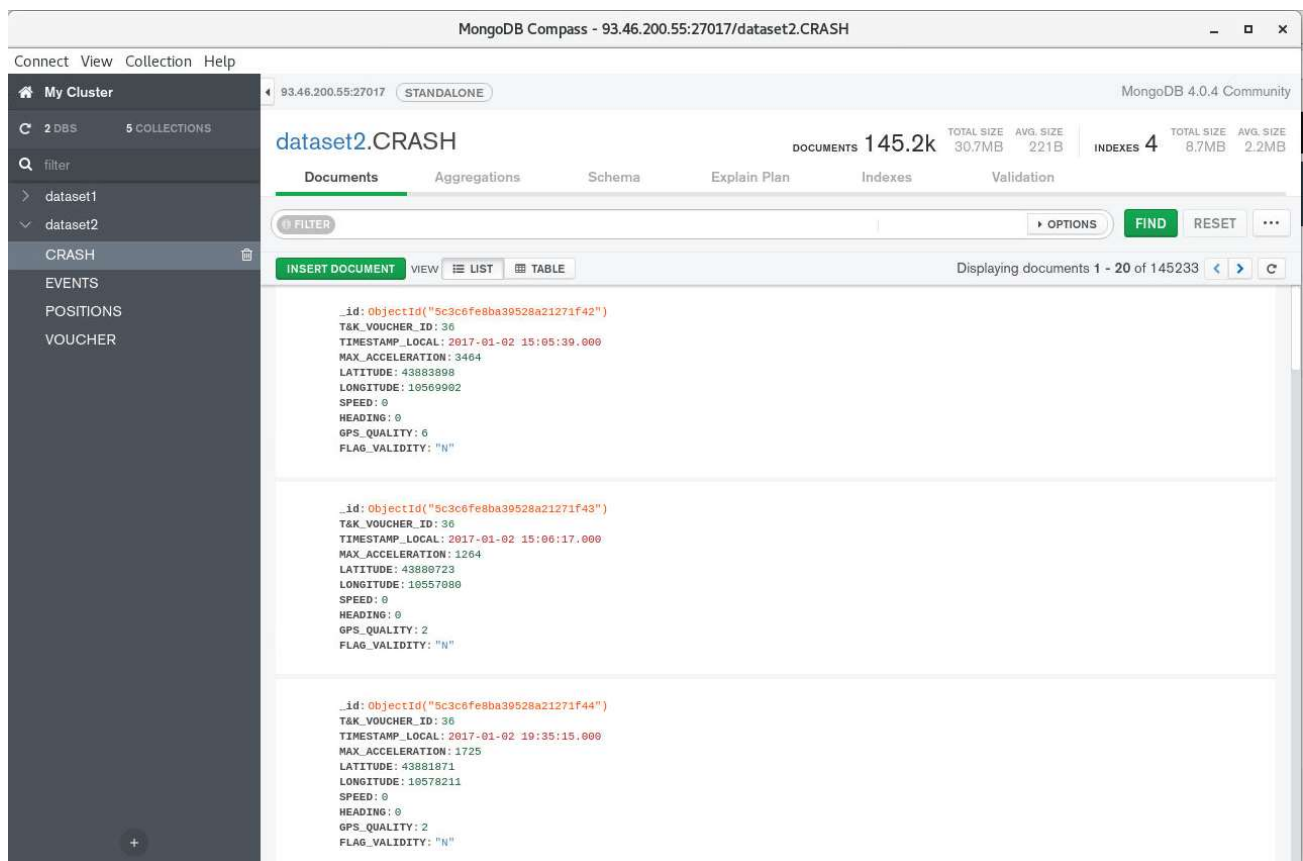


Figure 5.35 SIS CRASH collection

The available installed connectors can be accessed by pressing the “NEW” button in the Kafka Connect UI. After Selecting a particular connector (Here the SisCrashSourceConnector) the configuration has to be entered.

D2.3 Development of Toolboxes Integration Connectors

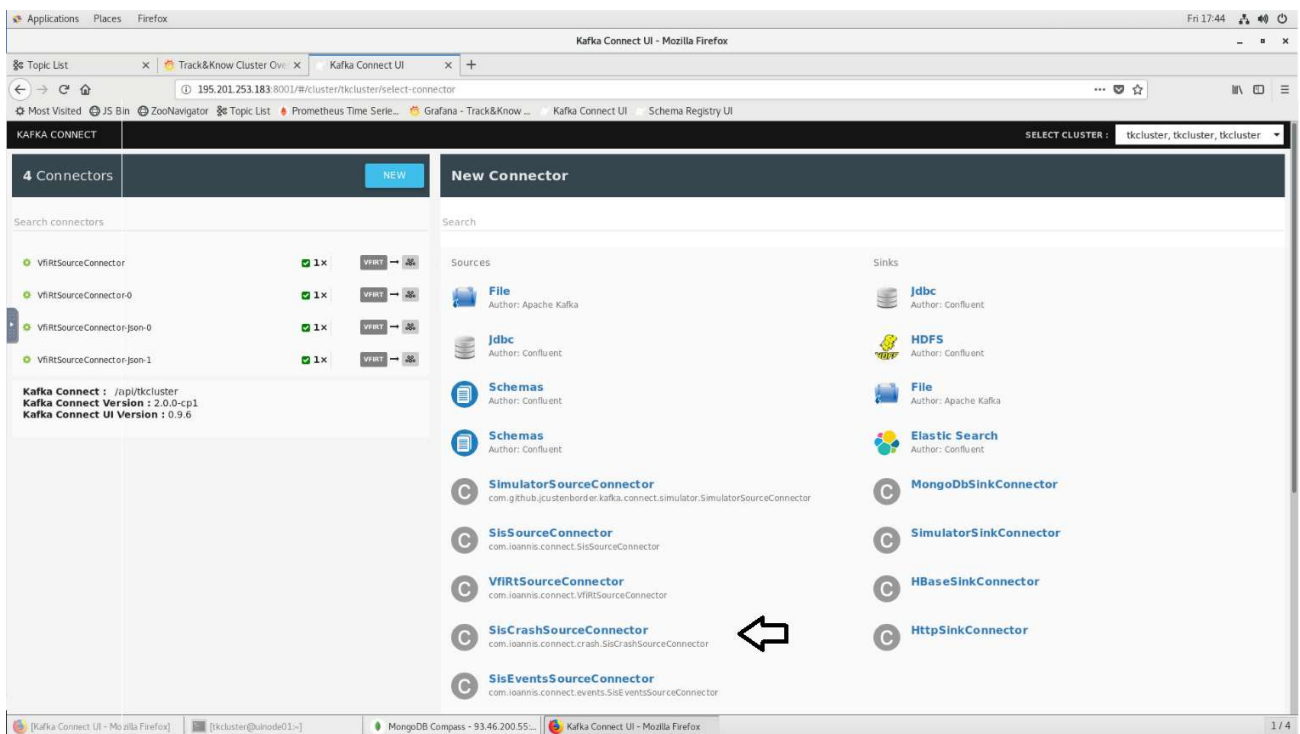


Figure 5.36 Selecting the CRASH connector

In the figure below the SisCrashSourceConnector has been selected and its configuration window is shown:

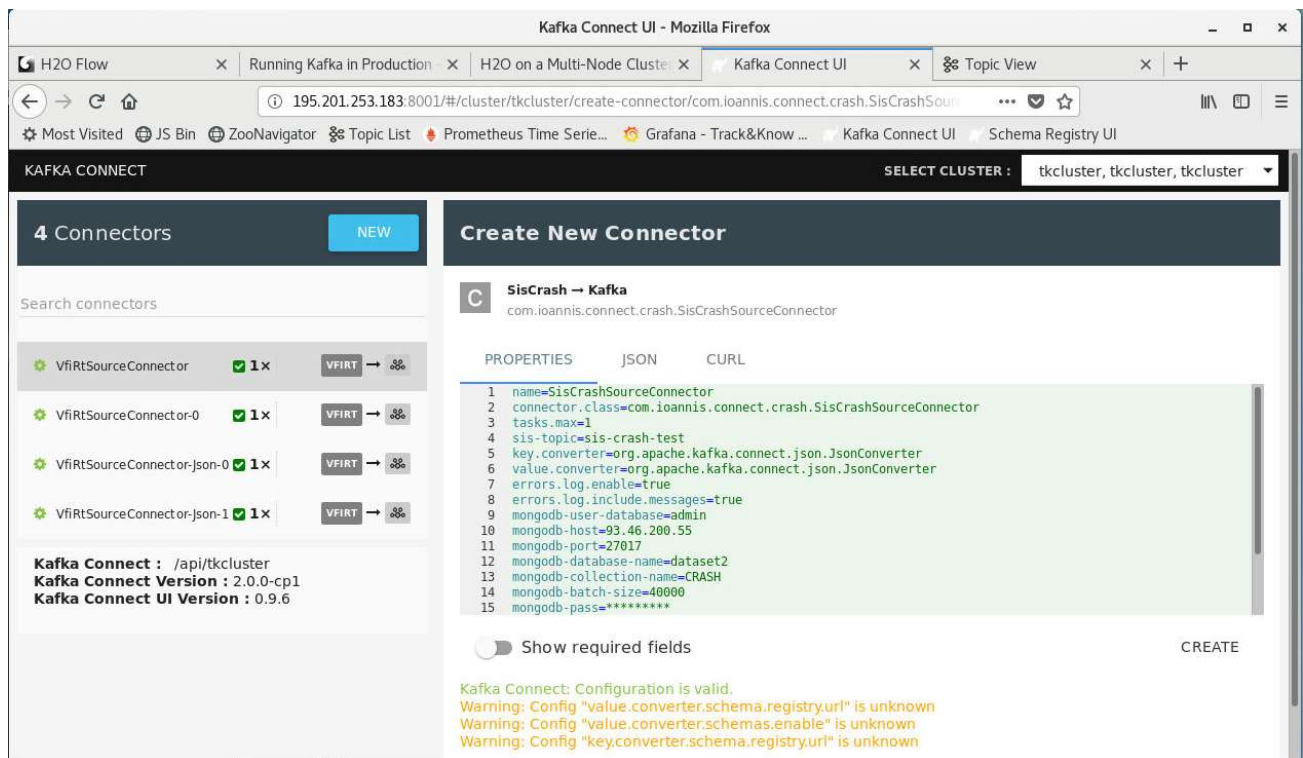


Figure 5.37 Configuring the CRASH connector

In detail, the configuration options are listed in the following table together with their description:

Table 7 SIS CRASH connector configuration options

Property and Sample value (Example)	Description
name=SisCrashSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.crash.SisCrashSourceConnector	The type of connector which is actually the connector's java class
tasks.max=1	The max number of tasks this connector should start.
sis-topic=d23-sis-crash	The topic where the retrieved data will be written in Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message key (here JSON is used)
value.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message value (here JSON is used)
errors.log.enable=true	Enables or disables error logging
errors.log.include.messages=true	Controls whether the error messages are included in the logs
mongodb-user-database=admin	The database collection to authenticate against
mongodb-host=93.46.200.55	The host IP where the MongoDB instance is running
mongodb-port=27017	The host port where the MongoDB instance is running
mongodb-database-name=dataset2	The database name where the collection is
mongodb-collection-name=CRASH	The name of the collection to be read
mongodb-batch-size=40000	The batch size that will be used
mongodb-pass=*****	The password to access the database
kafka-send-every-num-of-records=40000	The number of records that when gathered will be sent to Kafka.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the value schema registry
value.converter.schemas.enable=false	Setting to control whether the schema should also be sent

key.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081

The url for the key schema registry

5.2.3 Kafka Connect type Connector for the SIS EVENTS data

Similarly, data provided by SIS which contain information about events of interest that occurred and are related to the dataset were also made available in the same MongoDB instance setup in SIS premises and accessed over the network with the provided credentials. The database and related collection are named dataset2 and EVENTS respectively. A small portion of the data can be seen in the screenshot below taken from the MongoDB Compass Client while the latter is connected to the remote database and collection:

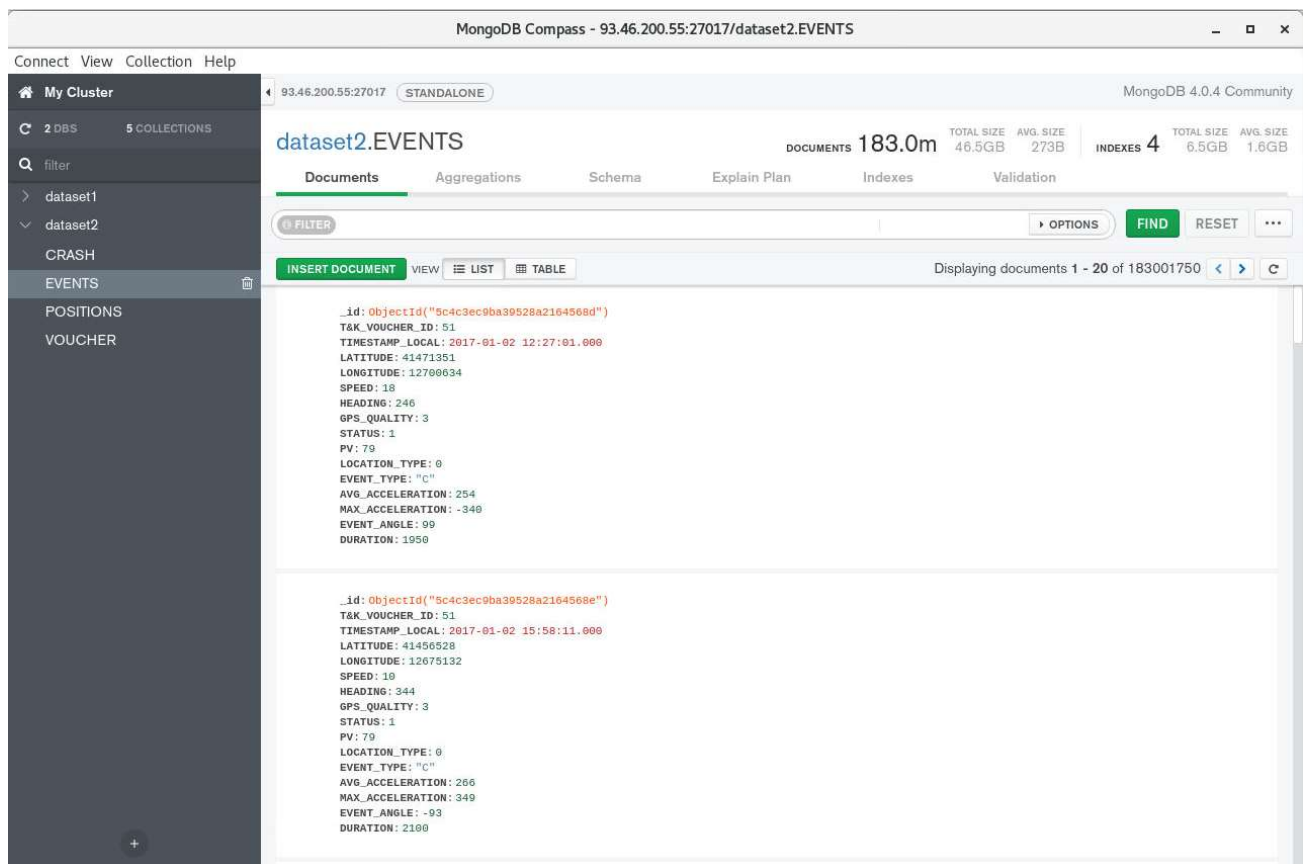


Figure 5.38 SIS EVENTS collection

The available installed connectors can be accessed by pressing the “NEW” button in the Kafka Connect UI. After selecting a particular connector (Here the SisEventsSourceConnector) the configuration has to be entered.

D2.3 Development of Toolboxes Integration Connectors

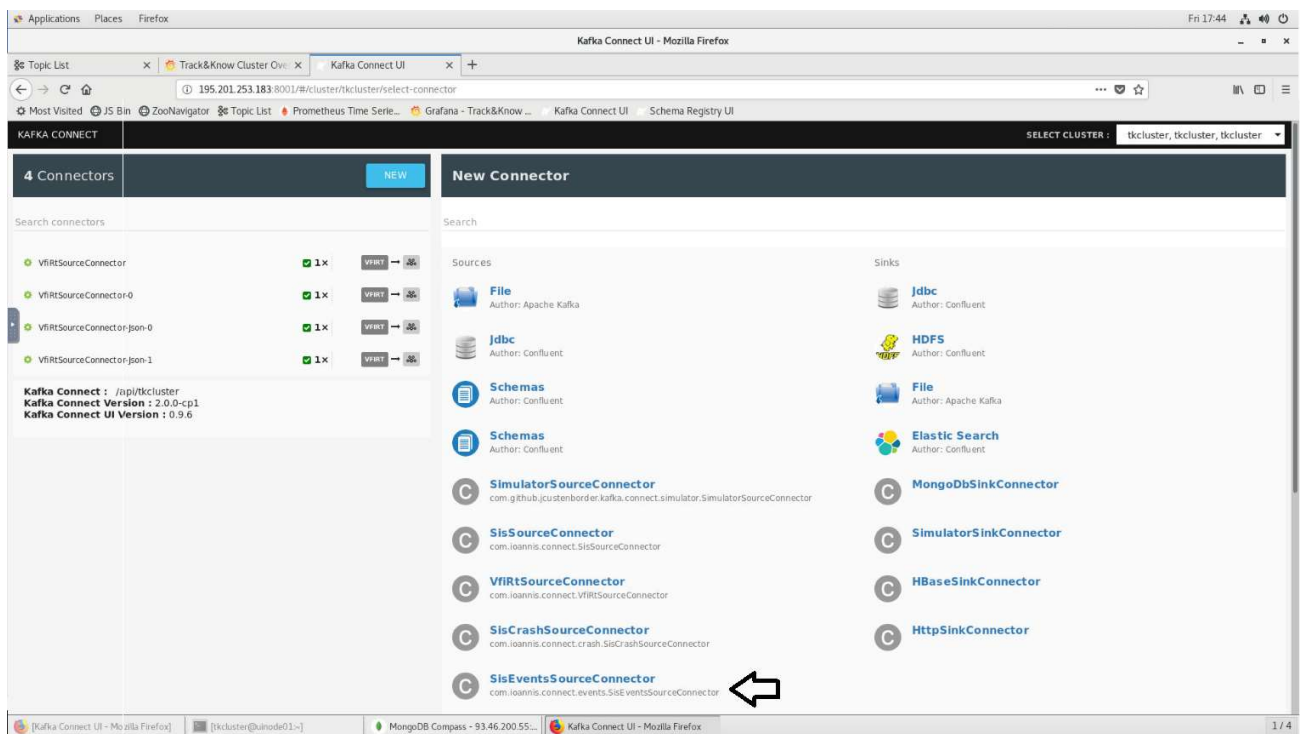


Figure 5.39 Selecting the EVENTS connector

In the figure below the SisEventsSourceConnector has been selected and its configuration window is shown:

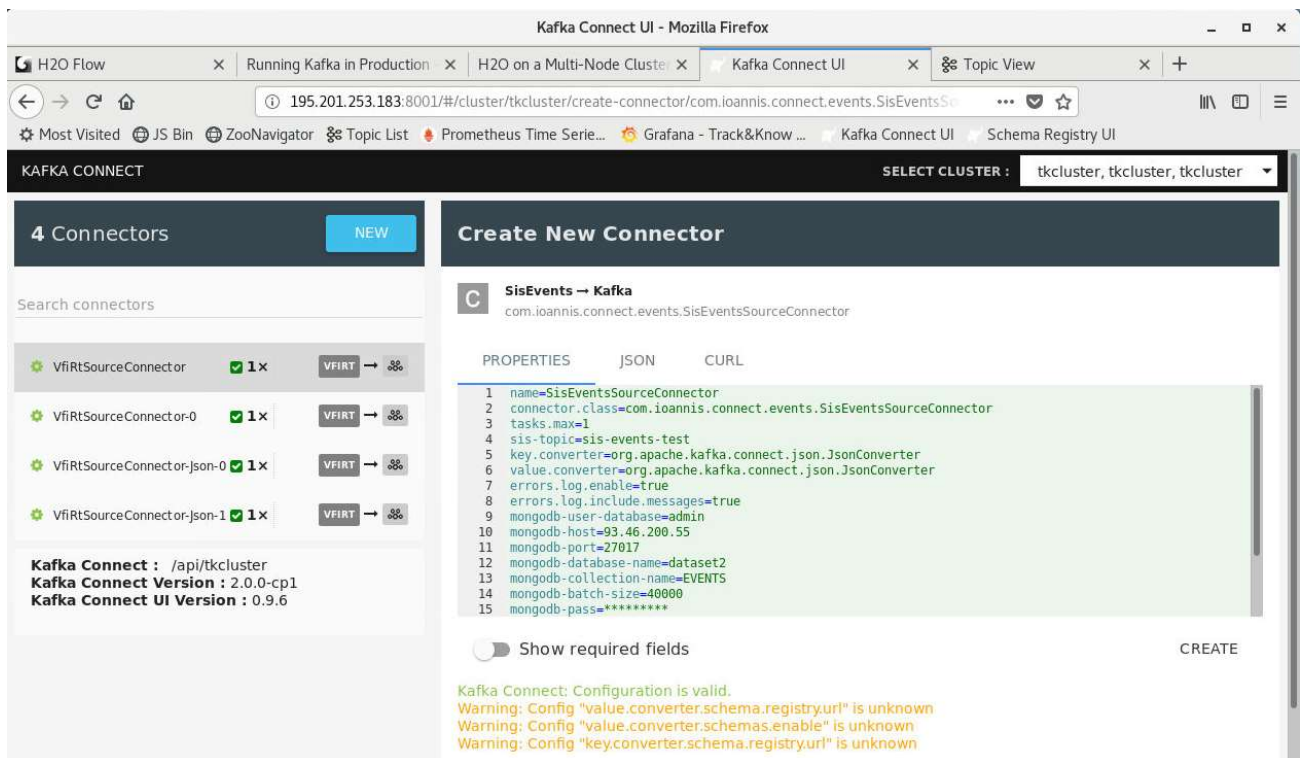


Figure 5.40 Configuring the EVENTS connector

In detail, the configuration options are listed in the following table together with their description:

Table 8 SIS EVENTS connector configuration options

Property and Sample value (Example)	Description
name=SisEventsSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.crash.SisEventsSourceConnector	The type of connector which is actually the connector's java class
tasks.max=1	The max number of tasks this connector should start.
sis-topic=d23-sis-events	The topic where the retrieved data will be written in Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message key (here JSON is used)
value.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message value (here JSON is used)
errors.log.enable=true	Enables or disables error logging
errors.log.include.messages=true	Controls whether the error messages are included in the logs
mongodb-user-database=admin	The database collection to authenticate against
mongodb-host=93.46.200.55	The host IP where the MongoDB instance is running
mongodb-port=27017	The host port where the MongoDB instance is running
mongodb-database-name=dataset2	The database name where the collection is in
mongodb-collection-name=EVENTS	The name of the collection to be read
mongodb-batch-size=40000	The batch size that will be used
mongodb-pass=*****	The password to access the database
kafka-send-every-num-of-records=40000	The number of records that when gathered will be sent to Kafka.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the value schema registry
value.converter.schemas.enable=false	Setting to control whether the schema should also be sent

key.converter.schema.registry.url=http://static.
185.253.201.195.clients.your-server.de:8081

The url for the key schema registry

5.2.4 Kafka Connect type Connector for the SIS POSITIONS data

In the same way, data provided by SIS which contain information about positions of interest that are related to the dataset were also made available in the same MongoDB instance setup in SIS premises and accessed over the network with the provided credentials. The database and related collection are named dataset2 and POSITIONS respectively. A small portion of the data can be seen in the screenshot below taken from the MongoDB Compass Client while the latter is connected to the remote database and collection:

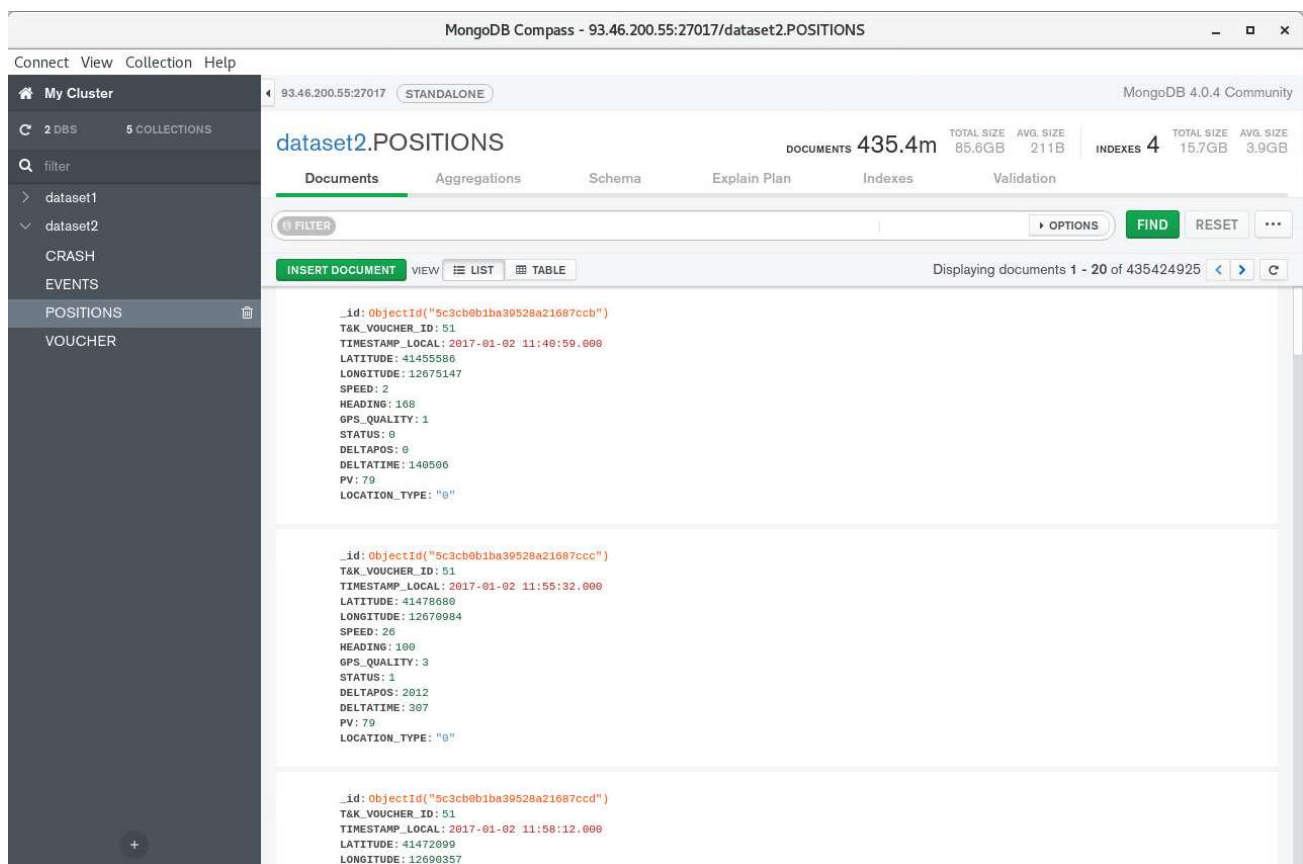


Figure 5.41 SIS POSITIONS collection

For the purpose of starting a SIS Kafka Connect type of connector, the Kafka Connect UI component is used all the connectors inside the Workers directory and the running connectors can be seen:

The available installed connectors can be accessed by pressing the “NEW” button in the Kafka Connect UI. After selecting a particular connector (Here the `SisPositionsSourceConnector`) the configuration has to be entered.

D2.3 Development of Toolboxes Integration Connectors

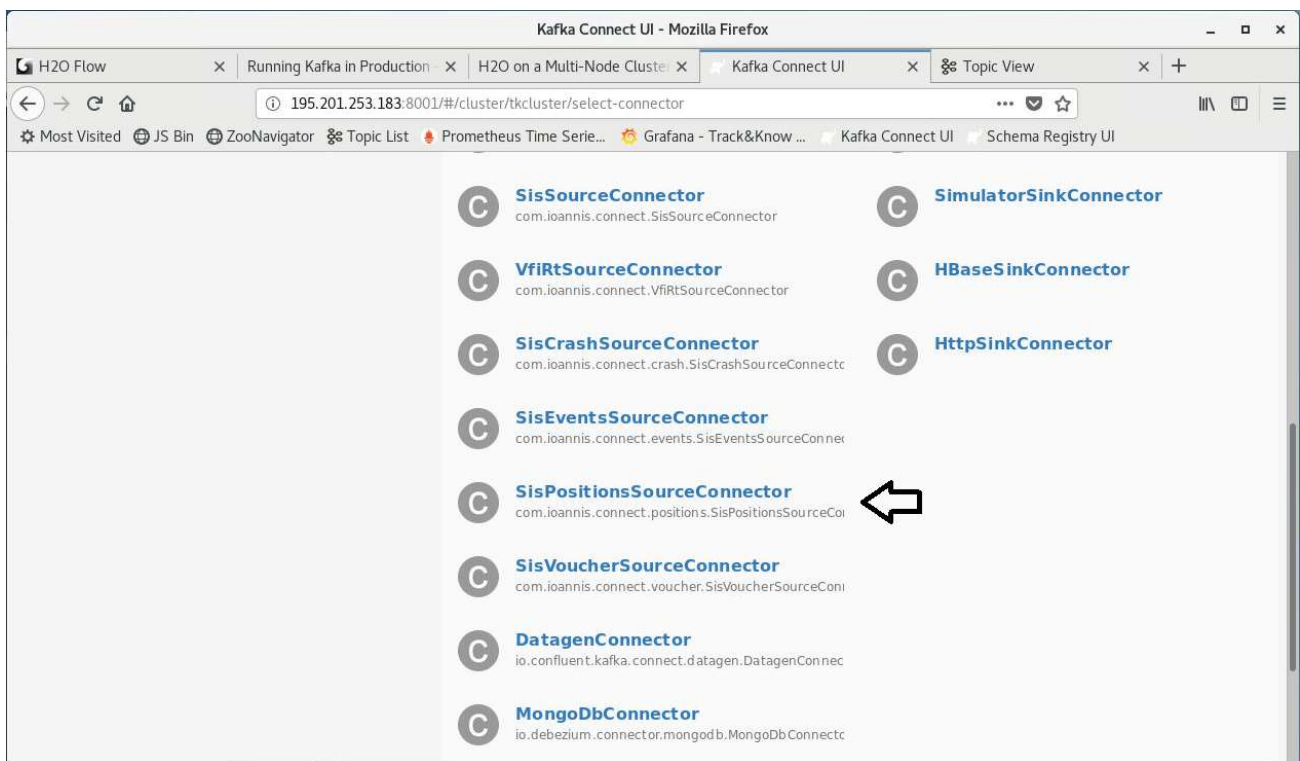


Figure 5.42 Selecting the POSITIONS connector

In the figure below the SisPositionsSourceConnector has been selected and its configuration window is shown:

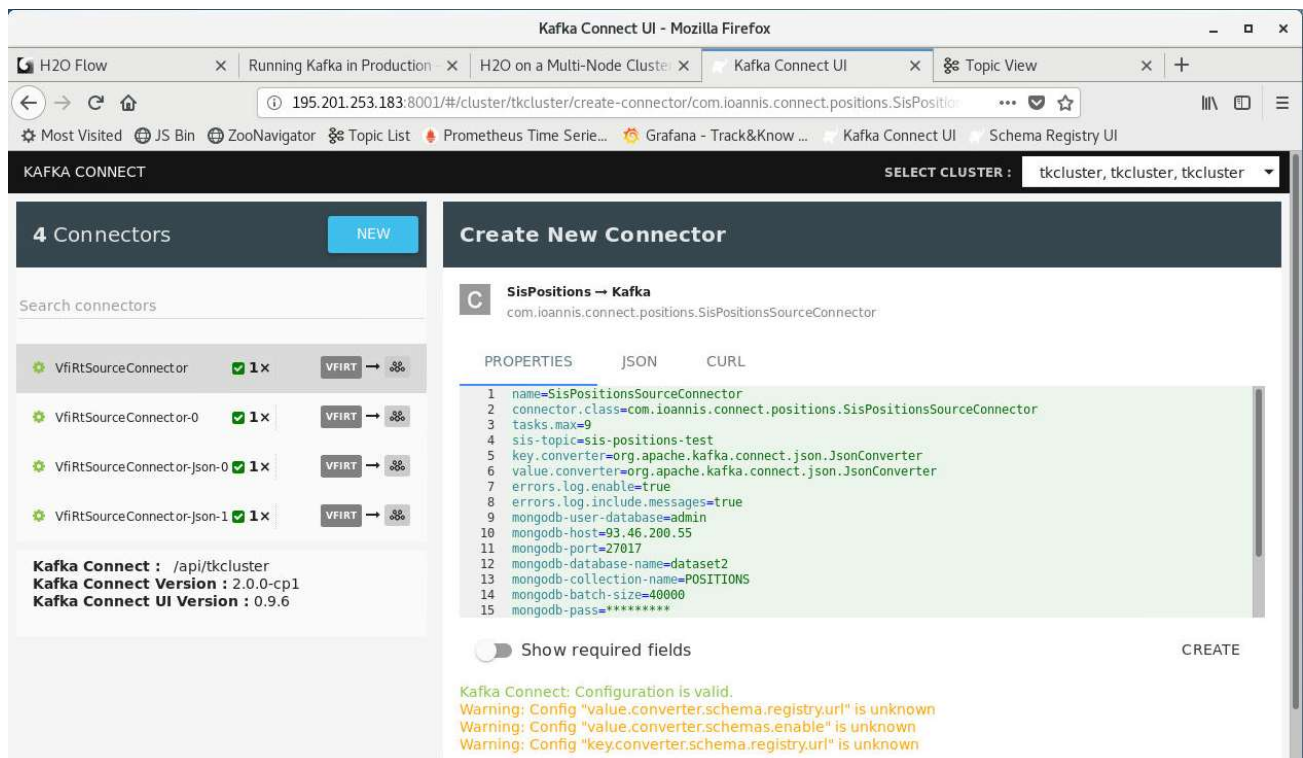


Figure 5.43 Configuring the POSITIONS connector

In detail, the configuration options are listed in the following table together with their description:

Table 9 SIS POSITIONS connector configuration options

Property and Sample value (Example)	Description
name=SisPositionsSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.positions.SisEventsPositionsConnector	The type of connector which is actually the connector's java class
tasks.max=1	The max number of tasks this connector should start.
sis-topic=d23-sis-positions	The topic where the retrieved data will be written in Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message key (here JSON is used)
value.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message value (here JSON is used)
errors.log.enable=true	Enables or disables error logging
errors.log.include.messages=true	Controls whether the error messages are included in the logs
mongodb-user-database=admin	The database collection to authenticate against
mongodb-host=93.46.200.55	The host IP where the MongoDB instance is running
mongodb-port=27017	The host port where the MongoDB instance is running
mongodb-database-name=dataset2	The database name where the collection is in
mongodb-collection-name=POSITIONS	The name of the collection to be read
mongodb-batch-size=40000	The batch size that will be used
mongodb-pass=*****	The password to access the database
kafka-send-every-num-of-records=40000	The number of records that when gathered will be sent to Kafka.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the value schema registry
value.converter.schemas.enable=false	Setting to control whether the schema should also be sent


```
key.converter.schema.registry.url=http://static.
185.253.201.195.clients.your-server.de:8081
```

The url for the key schema registry

5.2.5 Kafka Connect type Connector for the SIS VOUCHER data

Information related to the insurance voucher activation and the vehicles themselves was also provided by SIS and were made available in the same MongoDB instance setup in SIS premises and accessed over the network with the provided credentials. The database and related collection are named dataset2 and VOUCHER respectively. A small portion of the data can be seen in the screenshot below taken from the MongoDB Compass Client while the latter is connected to the remote database and collection:

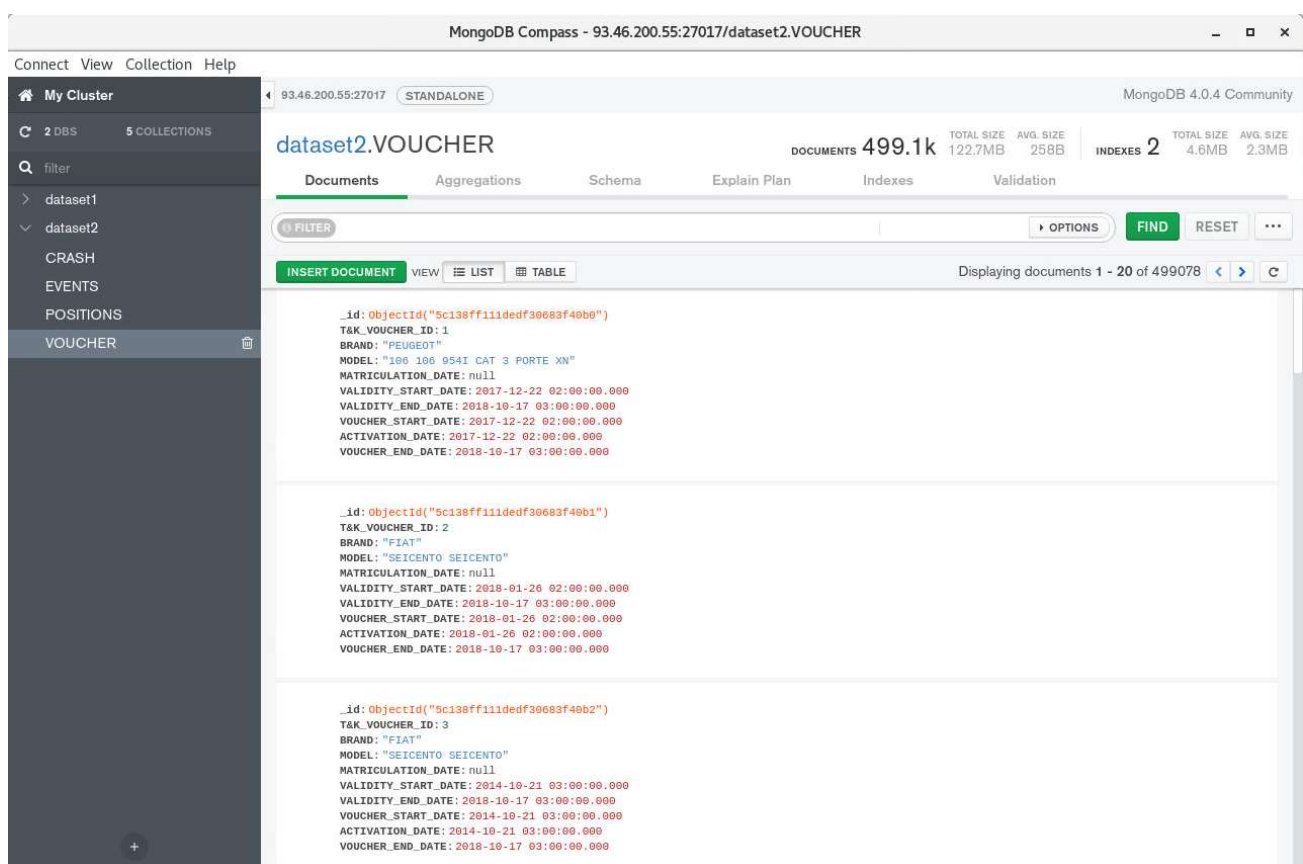


Figure 5.44 SIS VOUCHER collection

The available installed connectors can be accessed by pressing the “NEW” button in the Kafka Connect UI. After selecting a particular connector (Here the SisVoucherSourceConnector) the configuration has to be entered.

D2.3 Development of Toolboxes Integration Connectors

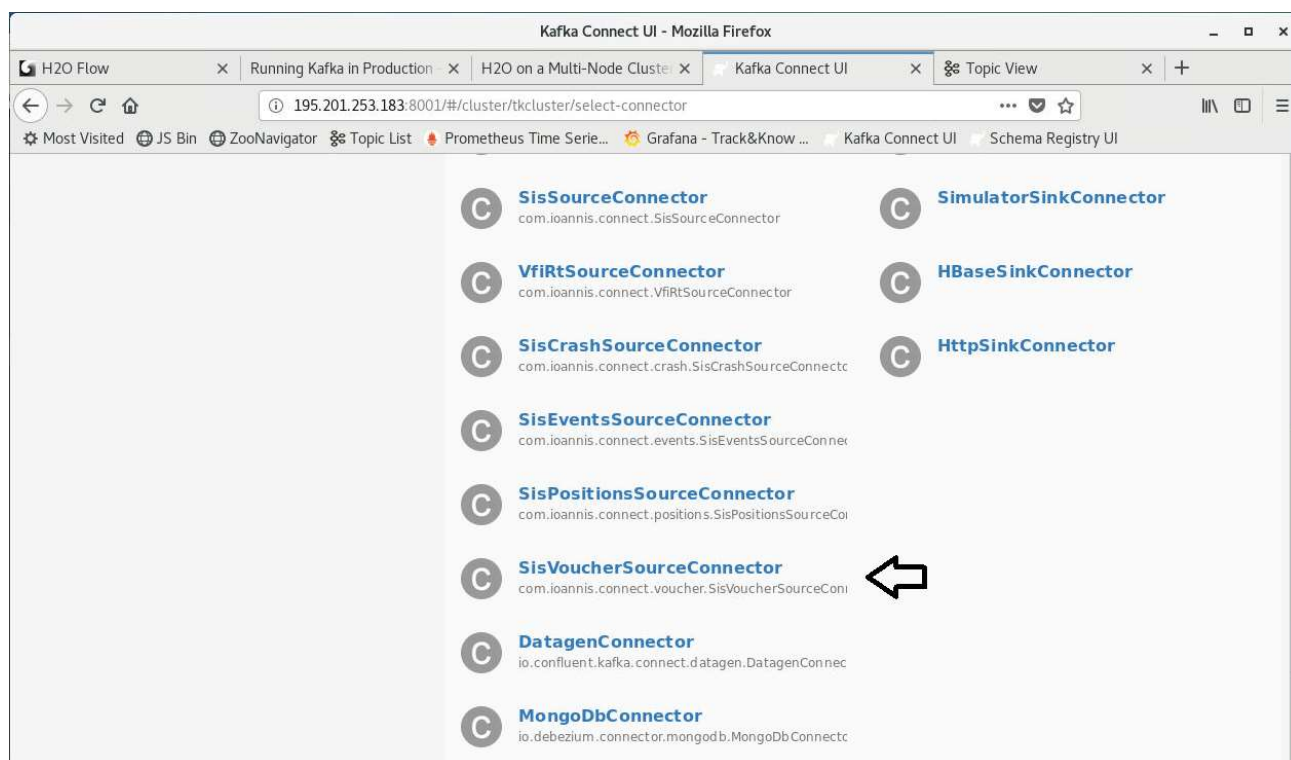


Figure 5.45 Selecting the VOUCHER connector

In the figure below the SisVoucherSourceConnector has been selected and its configuration window is shown:

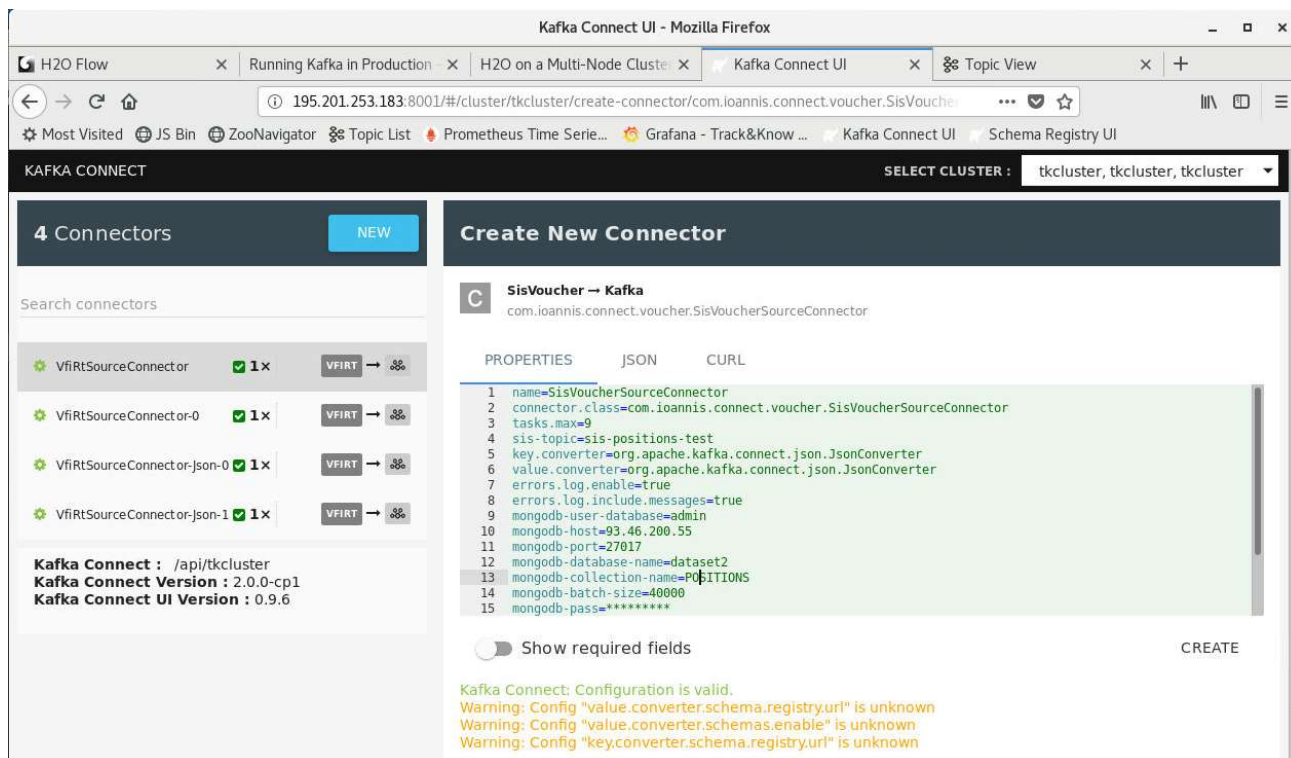


Figure 5.46 Configuring the VOUCHER connector

In detail, the configuration options are listed in the following table together with their description:

Table 10 SIS VOUCHER connector configuration options

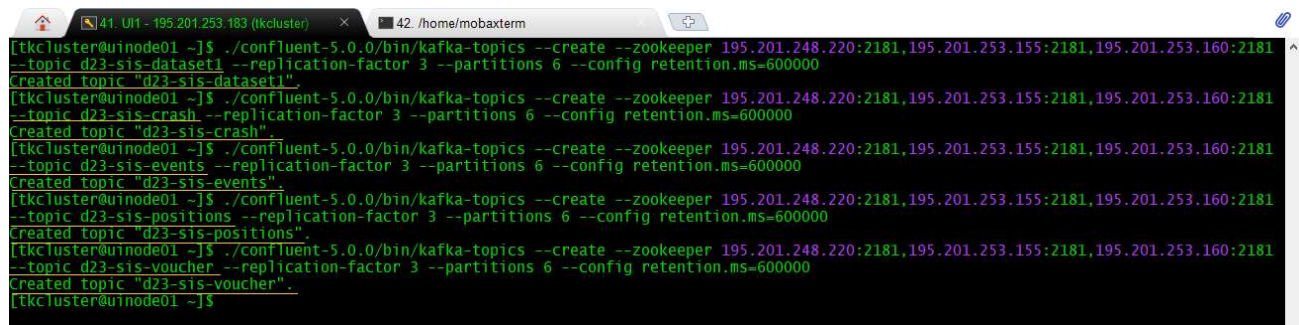
Property and Sample value (Example)	Description
name=SisVoucherSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.voucher.SisVoucherSourceConnector	The type of connector which is actually the connector's java class
tasks.max=1	The max number of tasks this connector should start.
sis-topic=d23-sis-voucher	The topic where the retrieved data will be written in Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message key (here JSON is used)
value.converter=org.apache.kafka.connect.json.JsonConverter	The converter used for the message value (here JSON is used)
errors.log.enable=true	Enables or disables error logging
errors.log.include.messages=true	Controls whether the error messages are included in the logs
mongodb-user-database=admin	The database collection to authenticate against
mongodb-host=93.46.200.55	The host IP where the MongoDB instance is running
mongodb-port=27017	The host port where the MongoDB instance is running
mongodb-database-name=dataset2	The database name where the collection is in
mongodb-collection-name=VOUCHER	The name of the collection to be read
mongodb-batch-size=40000	The batch size that will be used
mongodb-pass=*****	The password to access the database
kafka-send-every-num-of-records=40000	The number of records that when gathered will be sent to Kafka.
value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the value schema registry
value.converter.schemas.enable=false	Setting to control whether the schema should also be sent

key.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The url for the key schema registry
---	-------------------------------------

5.2.6 Introducing SIS datasets to the Track&Know Platform

By using the instructions presented in the previous sections of this chapter it is possible to start the SIS connectors and begin introducing data to the Kafka topics of choice. The similarities in the configuration and the actual SIS connectors presented are mainly due to the fact that the underlying task is extracting data from MongoDB and subsequently loading it into Kafka Topics. The main differences reside in the queries that each connector performs against the collection it is using and the actual objects employed to serialise and transmit the data to Kafka according to the Kafka Connect paradigm, which are interdependent to the different schemas applicable to each dataset in the MongoDB.

Before starting a connector, it is always advisable to create the topic that the data will be loaded. The figure below shows the creation of topics which are used to demonstrate the connectors. Specifically, for the SIS datasets a retention period must be set which makes the data vanish after a configurable time interval. For the examples below a 10 minute (600000 milliseconds) retention period is set:



```

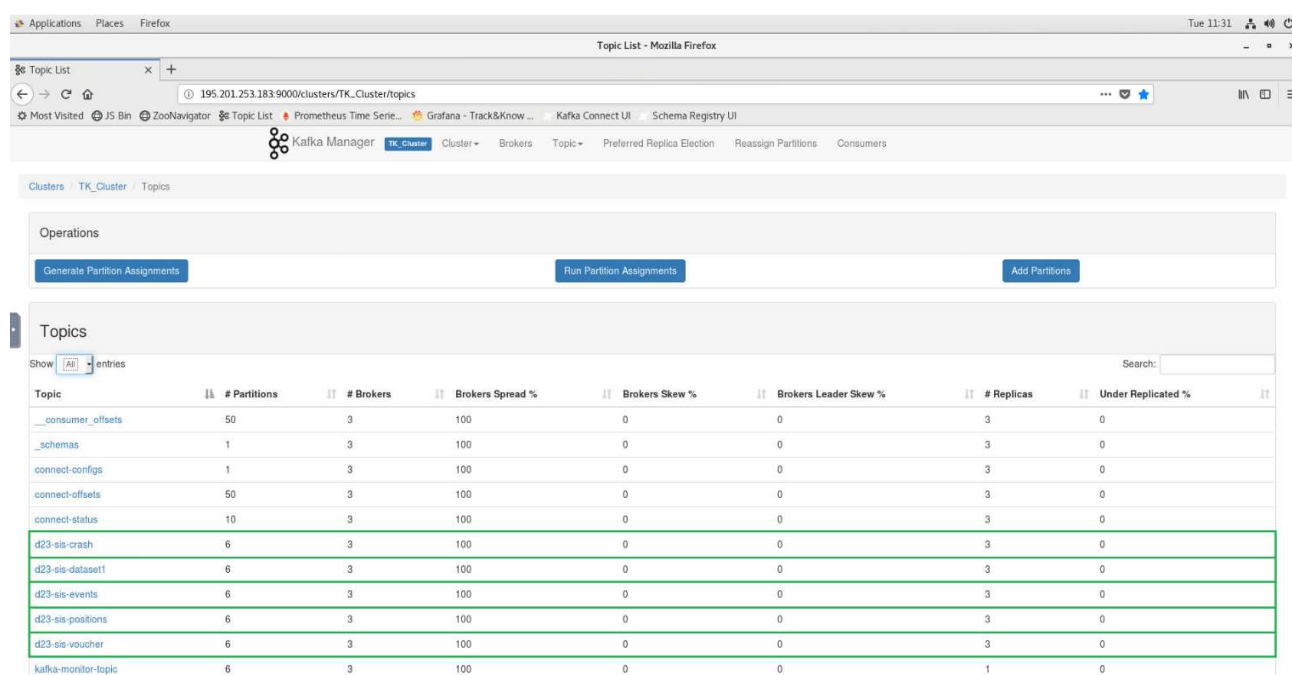
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-sis-dataset1 --replication-factor 3 --partitions 6 --config retention.ms=600000
Created topic "d23-sis-dataset1".
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-sis-crash --replication-factor 3 --partitions 6 --config retention.ms=600000
Created topic "d23-sis-crash".
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-sis-events --replication-factor 3 --partitions 6 --config retention.ms=600000
Created topic "d23-sis-events".
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-sis-positions --replication-factor 3 --partitions 6 --config retention.ms=600000
Created topic "d23-sis-positions".
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-sis-voucher --replication-factor 3 --partitions 6 --config retention.ms=600000
Created topic "d23-sis-voucher".
[tkcluster@uinode01 ~]$

```

Figure 5.47 Creating the SIS topics

The topics just created can be seen in Kafka Manager as discussed in earlier sections of the document:

D2.3 Development of Toolboxes Integration Connectors



Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %
_consumer_offsets	50	3	100	0	0	3	0
_schemas	1	3	100	0	0	3	0
connect-config	1	3	100	0	0	3	0
connect-offsets	50	3	100	0	0	3	0
connect-status	10	3	100	0	0	3	0
d23-sis-crash	6	3	100	0	0	3	0
d23-sis-dataset1	6	3	100	0	0	3	0
d23-sis-events	6	3	100	0	0	3	0
d23-sis-positions	6	3	100	0	0	3	0
d23-sis-voucher	6	3	100	0	0	3	0
kafka-monitor-topic	6	3	100	0	0	1	0

Figure 5.48 SIS topics seen in the Kafka Manager

In the figure below the started connectors can be seen:

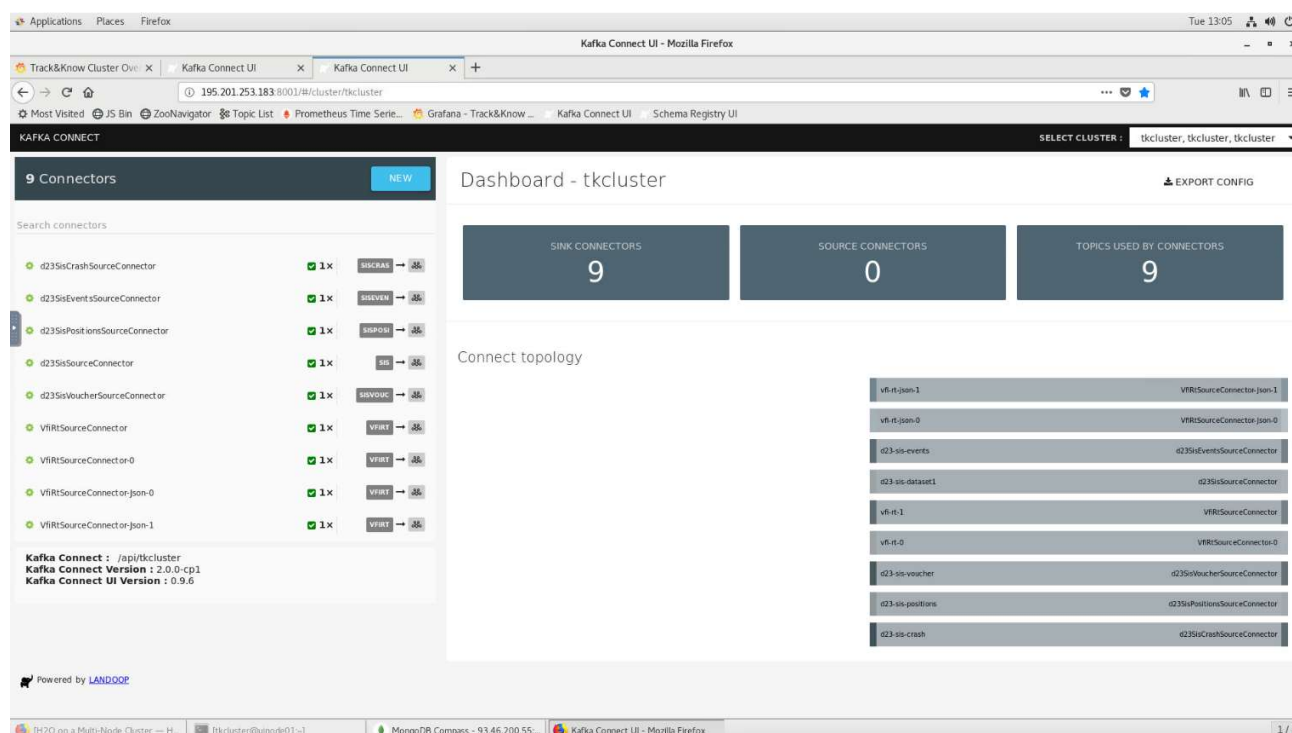
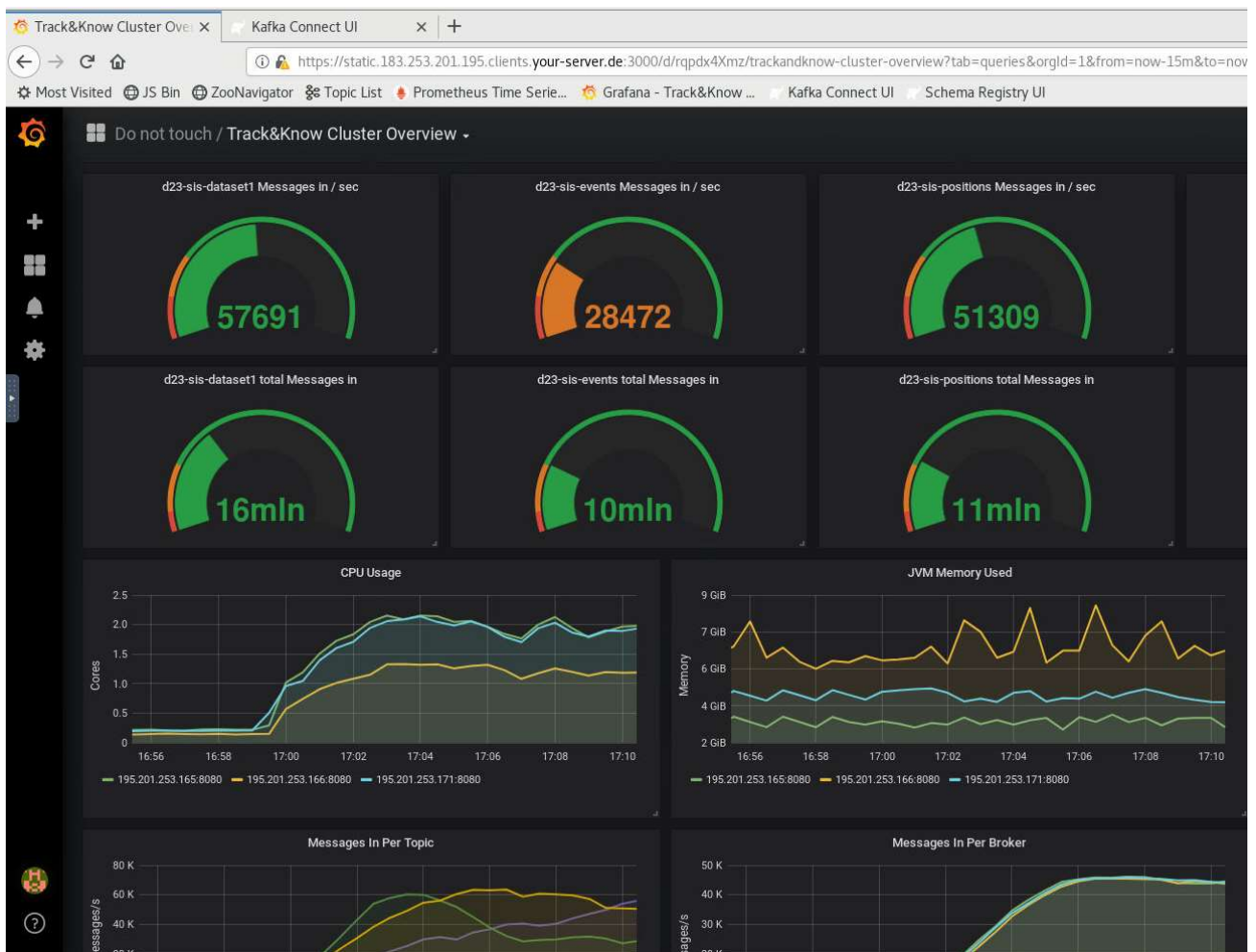


Figure 5.49 SIS Connectors running

The performance of the running connectors can be monitored in the Dashboard:

Figure 5.50 SIS Connectors at work

D2.3 Development of Toolboxes Integration Connectors



The message rates shown above are indicative at the time of writing, while performance tuning of the MongoDB instance which makes the data available can further improve them as the Track&Know Platform can introduce data into topics at higher message rates. It should be mentioned that the connectors support parallelism, variable batch sizes and other performance configurations which facilitate performance tuning.

By using the command below, JSON messages can be retrieved (read) from the topic of choice (here d23-sis-dataset-1 – similar for other SIS topics) and displayed in the console. The JSON format of the original data is maintained:


```

[wp3user05@wp3node01 ss1]$ ~/confluent-5.0.0/bin/kafka-console-consumer --bootstrap-server static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093 --topic d23-sis-dataset1 --from-beginning --consumer.config ~/connectors/client.properties
{"_id":"5c6cb3eaba39528a21dafa78","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:25:10 EEST 2017","LATITUDE":42341095,"LONGITUDE":11617464,"SPEED":86,"HEADING":152,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":620}
{"_id":"5c6cb3eaba39528a21dafa75","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:28:09 EEST 2017","LATITUDE":42317832,"LONGITUDE":11652241,"SPEED":74,"HEADING":142,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":654}
{"_id":"5c6cb3eaba39528a21dafa74","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:27:39 EEST 2017","LATITUDE":42320926,"LONGITUDE":11645664,"SPEED":76,"HEADING":102,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":608}
{"_id":"5c6cb3eaba39528a21dafa73","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:27:10 EEST 2017","LATITUDE":42324046,"LONGITUDE":11639858,"SPEED":78,"HEADING":136,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":621}
{"_id":"5c6cb3eaba39528a21dafa72","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:26:40 EEST 2017","LATITUDE":42326610,"LONGITUDE":11633324,"SPEED":72,"HEADING":142,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":673}
{"_id":"5c6cb3eaba39528a21dafa71","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:26:10 EEST 2017","LATITUDE":42331352,"LONGITUDE":11628270,"SPEED":86,"HEADING":138,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":731}
{"_id":"5c6cb3eaba39528a21dafa70","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:25:40 EEST 2017","LATITUDE":42336234,"LONGITUDE":11622341,"SPEED":86,"HEADING":138,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":677}
{"_id":"5c6cb3eaba39528a21dafa6f","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:24:40 EEST 2017","LATITUDE":42345523,"LONGITUDE":11613118,"SPEED":68,"HEADING":116,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":647}
{"_id":"5c6cb3eaba39528a21dafa6e","VEHICLE_ID":1908339,"TIMESTAMP":"Mon Aug 14 15:23:40 EEST 2017","LATITUDE":42350861,"LONGITUDE":11699440,"SPEED":76,"HEADING":146,"GPS_QUALITY":3,"STATUS":1,"DELTAPOS":650}

```

Figure 5.51 SIS topic data sample

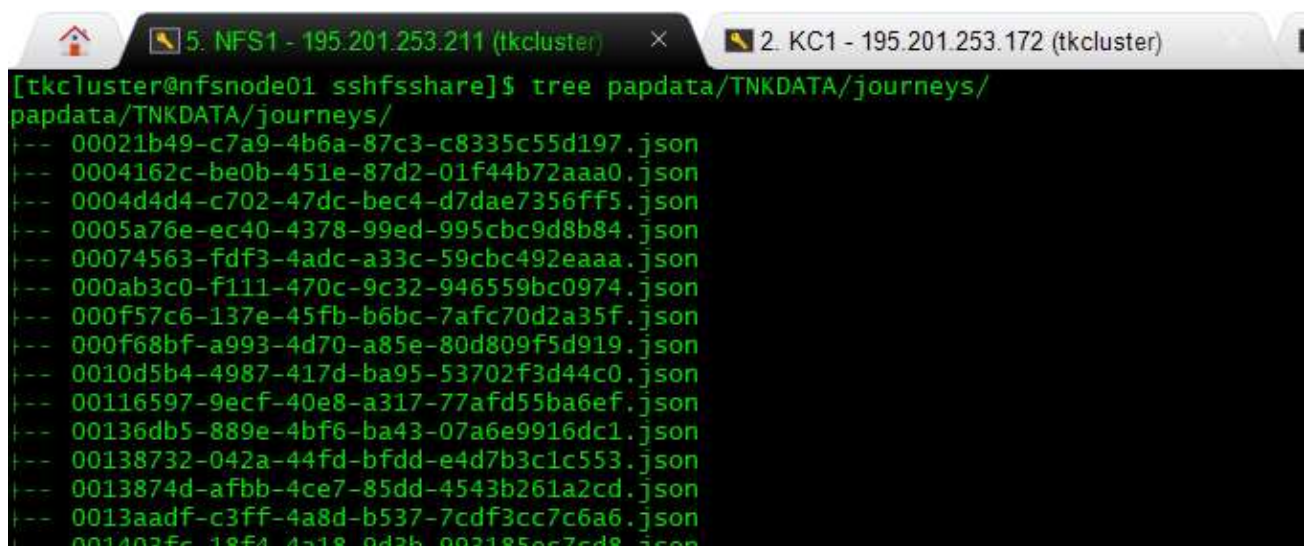
In addition to the above presented connectors there also exist Producer type of connectors for all the SIS data presented, which have been implemented and can be configured and run in a similar manner to what is described in section 5.1.1 of this document. Furthermore, for the better utilisation of the provided data by the other WPs it has been decided by SIS to produce a merged version of the datasets made available and presented in the above sections. The merged dataset produced unifies the information contained in the various collections presented above (except the VOUCHER information) in the METRICS Collection. The data is loaded by a single connector to the Track&Know Platform in a similar way to what has been already presented.

5.3 PAP Data Connectors

In this section the connectors implemented for introducing the PAP data to the Track&Know platform are discussed, presenting information about their usage, configuration, startup and monitoring.

5.3.1 Producer type Connector for the PAP reconstructed journey data

The reconstructed journey data contains information related to the patients' journeys from their home to the clinic by reconstructing GPS traces, directions, route timings and a poly line as a GeoJSON for each individual appointment from existing data. The dataset was made available in multiple JSON (.json) files, with the latter holding the related information. **Several .json files exist in the data directory which resides on an encrypted disk and is shared via SSHFS to other computing nodes (e.g. where a connector is run). All hosts in the platform feature encrypted drives and it is worth mentioning that all communications between Connectors, the Kafka Brokers, Toolbox code and clients in general are encrypted.** The files are organised in a way that can be seen in the figure below:



The screenshot shows a terminal window with two tabs. The active tab is titled '5. NFS1 - 195.201.253.211 (tkcluster)'. The command prompt is '[tkcluster@nfsnode01 sshfsshare]\$'. The command entered is 'tree papdata/TNKDATA/journeys/'. The output shows a list of JSON files with their full paths and names, including UUIDs and file extensions. The files are listed in a tree structure, with the root being 'papdata/TNKDATA/journeys/'.

```
[tkcluster@nfsnode01 sshfsshare]$ tree papdata/TNKDATA/journeys/
papdata/TNKDATA/journeys/
+-- 00021b49-c7a9-4b6a-87c3-c8335c55d197.json
+-- 0004162c-be0b-451e-87d2-01f44b72aaa0.json
+-- 0004d4d4-c702-47dc-bec4-d7dae7356ff5.json
+-- 0005a76e-ec40-4378-99ed-995cbc9d8b84.json
+-- 00074563-fdf3-4adc-a33c-59cbc492eaaa.json
+-- 000ab3c0-f111-470c-9c32-946559bc0974.json
+-- 000f57c6-137e-45fb-b6bc-7afc70d2a35f.json
+-- 000f68bf-a993-4d70-a85e-80d809f5d919.json
+-- 0010d5b4-4987-417d-ba95-53702f3d44c0.json
+-- 00116597-9ecf-40e8-a317-77afd55ba6ef.json
+-- 00136db5-889e-4bf6-ba43-07a6e9916dc1.json
+-- 00138732-042a-44fd-bfdd-e4d7b3c1c553.json
+-- 0013874d-afbb-4ce7-85dd-4543b261a2cd.json
+-- 0013aadf-c3ff-4a8d-b537-7cdf3cc7c6a6.json
+-- 001403fc-18f4-4a18-8d3b-802185ec7cd8.json
```

Figure 5.52 PAP journey data in JSON files

Due to the confidential nature of the information, actual screenshots of the data cannot be shown. For the purpose of documenting the structure of the JSON data, a figure with the values set to zero and omitted information is provided:

Figure 5.53 PAP journey data JSON sample

Taking in account the above, it was decided to create a connector that could (if needed) upload the data in parallel. The approach that was adopted resulted in a Producer that can utilise multiple threads per instance for loading the data and writing the messages to the Apache Kafka Cluster.

The Connector is packaged in a jar and can be built from the sources by using mvn clean, compile and assembly single as it can be seen below (IntelliJ IDE):

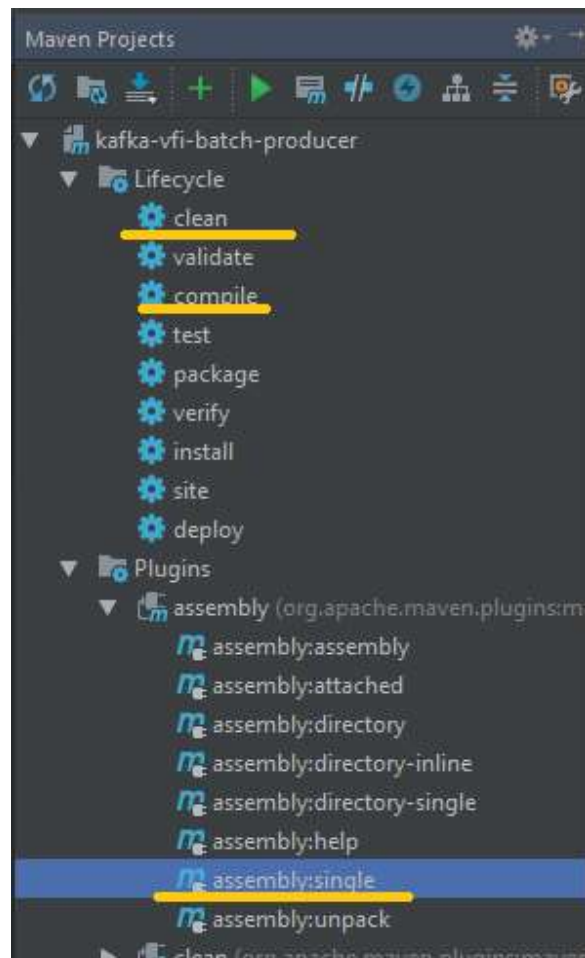


Figure 5.54 Building and assembling the PAP journey batch producer

The jar that is produced can then be copied to the VMs that the Connector will be run. Alternatively, the code can be pulled at a VM and the connector can be built there. Once the jar is built it should be made executable (sudo chmod +x) and placed in a folder of choice together with its property file:

```
[tkcluster@uinode01 papconnect]$ ll
total 10096
-rwxrwxr-x. 1 tkcluster tkcluster 10332188 May 15 18:52 kafka-pap-reconstructed-1.0-jar-with-dependencies.jar
-rw-rw-r--. 1 tkcluster tkcluster    935 May 15 17:26 papbatchproducer.properties
[tkcluster@uinode01 papconnect]$
```

Figure 5.55 Producer jar and properties file

The property file must have the name papbatchproducer.properties and exist in the same folder as the jar, otherwise it cannot be read and the connector will fail to start. The Connector uses encryption (TLSv1.2) and compression (Snappy algorithm) while sending data to Apache Kafka and offers configurable batching and exactly once delivery. These and other properties are configured in the properties file and are explained below in more detail (example shown):

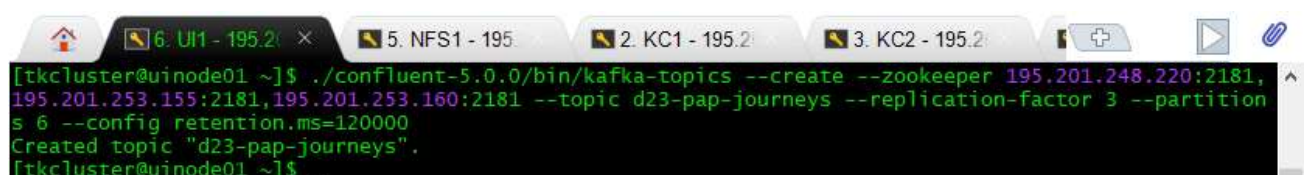
Table 11 Properties and sample values for the VFI batch producer

Property and Sample value (Example)	Description
topic=d23-pap-journeys	The Kafka topic that this connector will write data to
work.with.dir=/home/tkcluster/sshfsshare/pap data/journeys	The directory in which the batch connector will look for data. The data should be organised in the format discussed i.e. the folder should contain .json files.
worker.threads.count=8	The number of worker threads that this connector instance will employ when reading in files and writing data
producer.id=0	The id of this producer. If there are multiple then all must have different ids.
total.producers=1	The number of total producers. The loading task will be split among them and they will run in parallel.
key.serializer=org.apache.kafka.common.serialization.StringSerializer	The serializer used for the message key
value.serializer=org.apache.kafka.connect.json.JsonSerializer	The serializer used for the message value
bootstrap.servers=static.165.253.201.195.clients.your-server.de:9093,static.166.253.201.195.clients.your-server.de:9093,static.171.253.201.195.clients.your-server.de:9093	These are the 3 Kafka Brokers That the Connector will send data to. Note that port 9093 is used because only encrypted connections are allowed.
security.protocol=SSL	Authentication and encryption settings – Enables security.
ssl.enabled.protocols=TLSv1.2	Authentication and encryption settings – Enforces TLSv1.2
ssl.truststore.location=/home/wp3user04/ssl/kafka.client.truststore.jks	Authentication and encryption settings – Location of the Truststore that will be used to test Broker certificates.
ssl.truststore.password=password	Authentication and encryption settings – Truststore password
ssl.keystore.location=/home/wp3user04/ssl/kafka.client.keystore.jks	Authentication and encryption settings – Location of the Keystore where this Producer's certificate is stored.
ssl.keystore.password= password	Authentication and encryption settings – Keystore password

ssl.key.password= password	Authentication and encryption settings – Key password
enable.idempotence=true	Setting for idempotent producer. This will enable exactly once delivery.
acks=all	Setting required for idempotent producer. This will enable acks from all replicas.
retries=2147483647	Setting required for idempotent producer. This will enable max number of retries of failed messages.
max.in.flight.requests.per.connection=5	Setting required for idempotent producer. This will enable at most 5 unacknowledged messages while sending data.
compression.type=snappy	Compression algorithm used
linger.ms=5	Performance specific settings. Time to wait for a batch to fill. Optimal values may vary depending on cluster, message sizes etc.
batch.size=16384	Performance specific settings. Batch size. Optimal values may vary depending on cluster, message sizes etc.
request.timeout.ms=10000	Performance specific settings. Time to wait for a request to complete. Optimal values may vary depending on cluster, message sizes etc.

The above properties must be configured before the Connector can be started. Ensure that the topic is created beforehand and according to replication, partitioning, retention and size needs:

1. Please ensure that the topic where the data will be loaded has been created before starting the PAP/VFI Batch Producer Connector

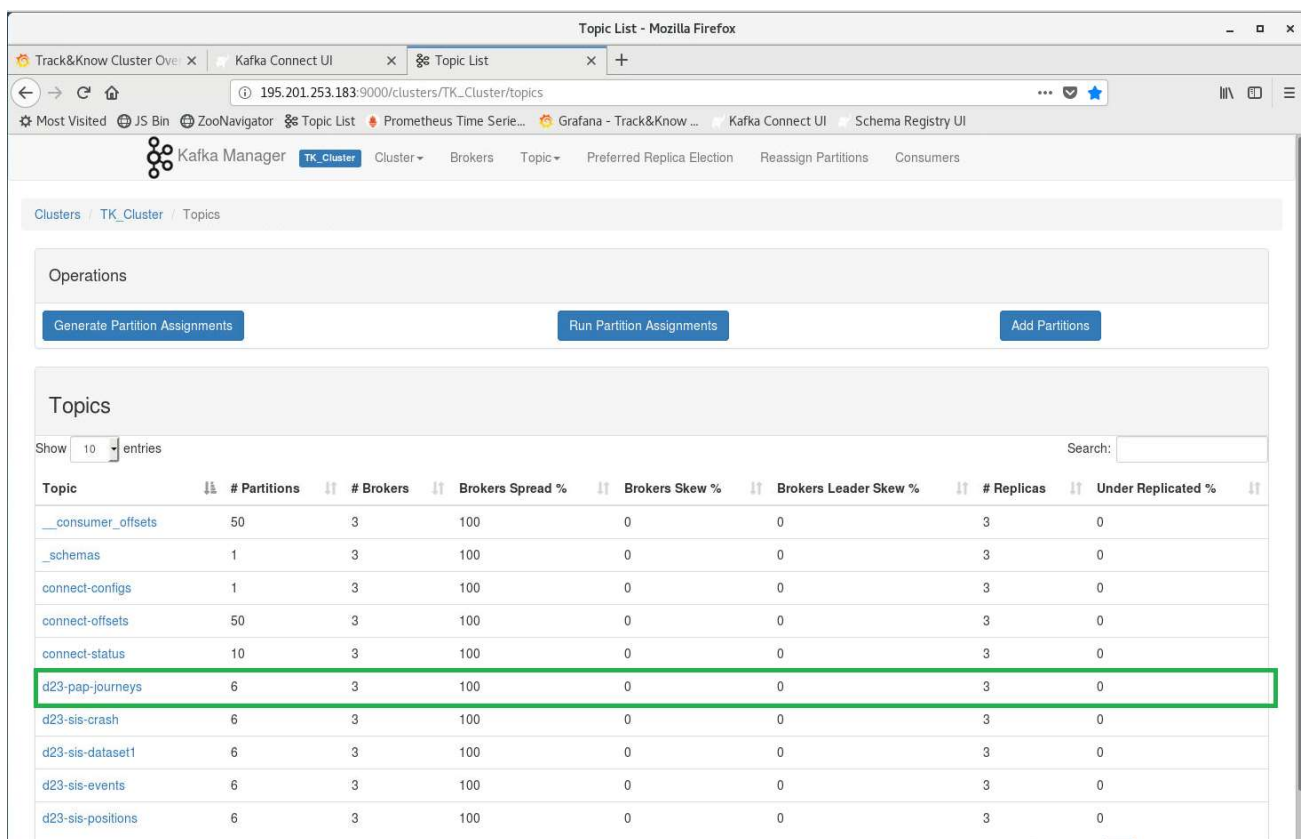


```
[tkcluster@uinode01 ~]$ ./confluent-5.0.0/bin/kafka-topics --create --zookeeper 195.201.248.220:2181,195.201.253.155:2181,195.201.253.160:2181 --topic d23-pap-journeys --replication-factor 3 --partitions 6 --config retention.ms=120000
Created topic "d23-pap-journeys".
[tkcluster@uinode01 ~]$
```

Figure 5.56 Topic creation for VFI/PAP journey data

The topic just created can be seen in Kafka Manager presented in earlier sections of the document:

D2.3 Development of Toolboxes Integration Connectors



Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %
__consumer_offsets	50	3	100	0	0	3	0
__schemas	1	3	100	0	0	3	0
connect-configs	1	3	100	0	0	3	0
connect-offsets	50	3	100	0	0	3	0
connect-status	10	3	100	0	0	3	0
d23-pap-journeys	6	3	100	0	0	3	0
d23-sis-crash	6	3	100	0	0	3	0
d23-sis-dataset1	6	3	100	0	0	3	0
d23-sis-events	6	3	100	0	0	3	0
d23-sis-positions	6	3	100	0	0	3	0

Figure 5.57 Topic for VFI/PAP data in Kafka Manager

2. At this point the `papbatchproducer.properties` file where the jar loads properties from should be edited. The topic just created should be set accordingly (in this example “d23-pap-journeys”). Also, the directory where the data to be loaded reside should be valid.
3. The Connector can be started by using the command below. The memory it is allowed to use can be increased for better performance.

```
[tkcluster@uinode01 ~]$ java -Xmx6G -jar kafka-pap-reconstructed-1.0-jar-with-dependencies.jar _
```

Figure 5.58 PAP/VFI batch connector startup command

Below a single connector can be seen while starting up:

```
[tkcluster@uinode01 papconnect]$ java -Xmx6G -jar kafka-pap-reconstructed-1.0-jar-with-dependencies.jar
May 16, 2019 12:49:51 PM com.ioannis.papproducers.reconstructeddataset.PapReconstructedBatchProducerSSL mai
n
INFO: >>> Starting up PAP Reconstructed Data Batch Connector
>>> Loading properties from /home/tkcluster/papconnect
-- listing properties --
max.in.flight.requests.per.connection=5
retries=2147483647
key.serializer=org.apache.kafka.common.serialization...
request.timeout.ms=180000
acks=all
ssl.keystore.location=/home/tkcluster/ssl/kafka.client.keys...
total.producers=1
ssl.enabled.protocols=TLSv1.2
ssl.truststore.location=/home/tkcluster/ssl/kafka.client.trus...
ssl.truststore.password=Tkc5^p
batch.size=327680
security.protocol=SSL
topic=d23-pap-journeys
worker.threads.count=10
enable.idempotence=true
ssl.keystore.password=Tkc5^p
linger.ms=5
bootstrap.servers=static.165.253.201.195.clients.your-s...
ssl.key.password=Tkc5^p
producer.id=0
value.serializer=org.apache.kafka.connect.json.JsonSer...
work.with.dir=/home/tkcluster/sshfsshare/papdata/TN...
compression.type=snappy
org.apache.kafka.connect.json.JsonSerializer
May 16, 2019 12:49:51 PM com.ioannis.papproducers.reconstructeddataset.PapReconstructedBatchProducerSSL mai
n
INFO: Total Journey Files: 45334
May 16, 2019 12:49:52 PM com.ioannis.papproducers.reconstructeddataset.PapReconstructedBatchProducerSSL mai
n
```

Figure 5.59 VFI/PAP journey connector startup output

4. Incoming message rates for the topic created can be observed in the Grafana Cluster Overview. The image below shows interesting information about the performance of the connector. Due to the small size of data in this experiment by the time that the incoming message rate climbs to 400 messages per second the total of 45334 journeys have been already loaded with total time taken to be around 60 seconds. The message rate would continue to climb with a large enough dataset and the expected rates can scale to the metrics presented in 5.1.1.

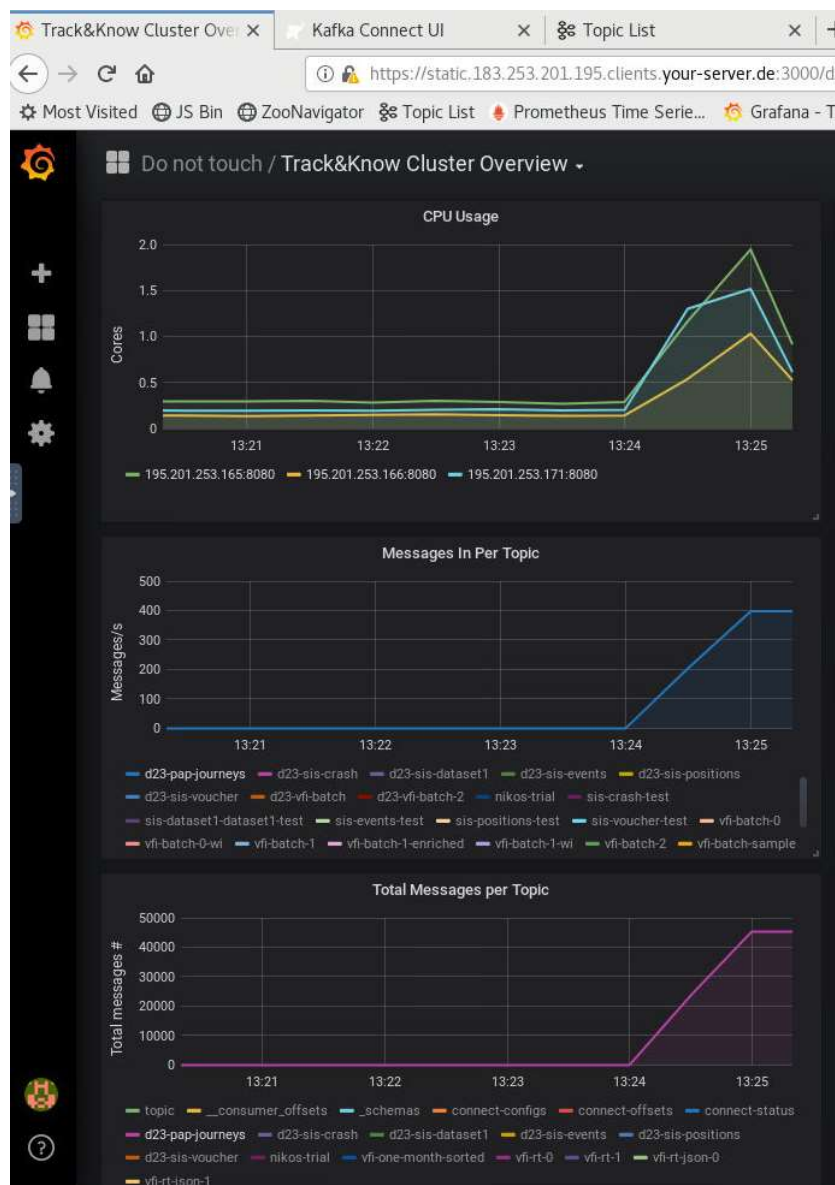


Figure 5.60 Incoming messages and rates for PAP journeys

5.3.2 Kafka Connect type Connector for the VFI/PAP Smartphone app live data

For introducing the PAP/VFI live data from the developed smartphone logger app to the Track&Know Platform, the Kafka Connect functionality was used. Again, a custom Kafka Connect Source Connector was developed to be deployed in the highly available Connect Cluster of the Track&Know Platform. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the PAP/VFI live app data and load them into the topic of choice, according to the connector code and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will proceed to run the connector code. This means that even at the case of failure of a node, the VFI/PAP live data will continue to enter the Platform. At configurable intervals the connector pulls the live data from a provided REST API and for a set number of vehicles, to be written to a Kafka topic of choice. The VFI API for the smartphone app makes available the data in JSON format, with the connector supporting serialisation in both JSON and Avro. Authentication and encryption settings are

already setup between the Kafka Connect Workers and the Kafka Brokers and no setup is necessary for individual connectors.

The connector and other required jars are built using mvn clean compile package which produces a target folder containing the jars. This folder is then moved to the dedicated directory of each Worker node of the Kafka Connect Cluster and this way the connector code is available and can be instantiated.

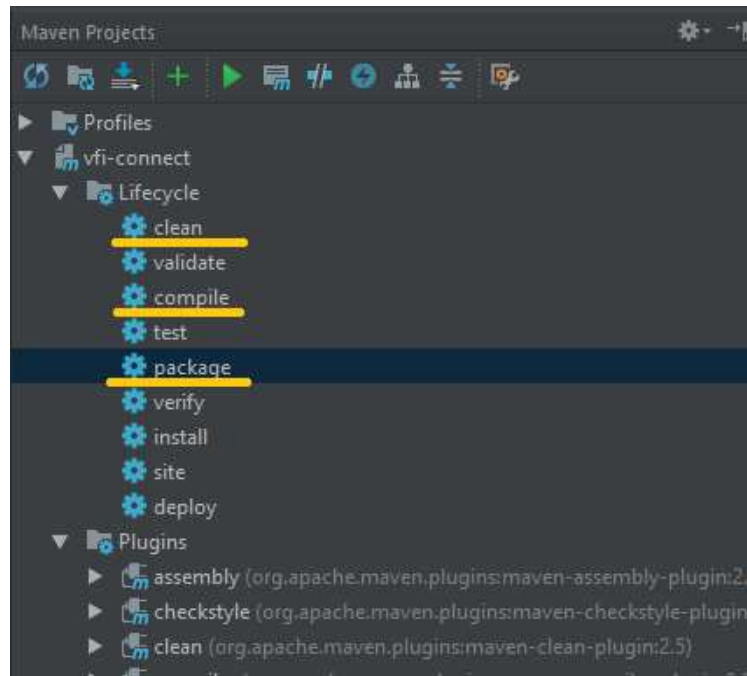


Figure 5.61 Building the pap-vfi-live connector

The contents of the connectors directory (here connect-jars) can be seen below. The PAP-VFI live data connector exists in the pap-vfi-live-connect directory:

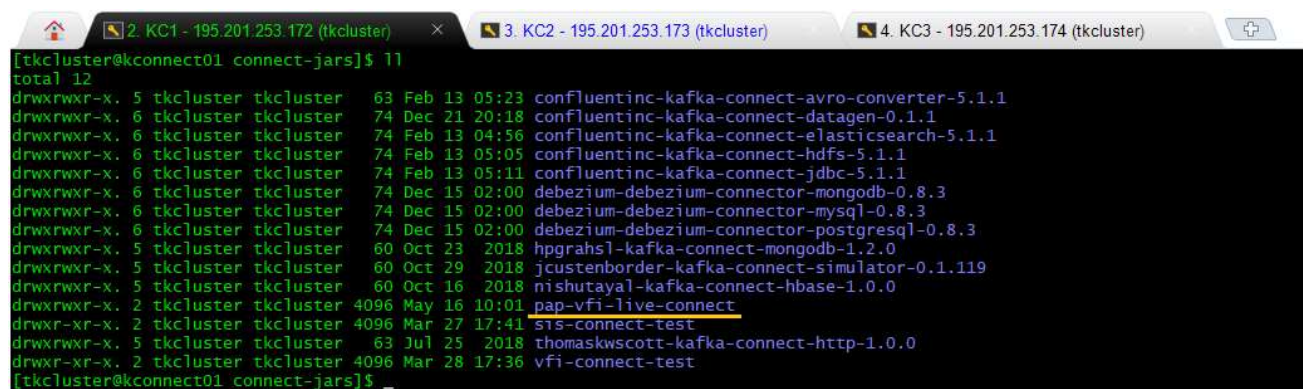


Figure 5.62 The pap-vfi connector dir.

For the purpose of starting the connector, the Kafka Connect UI component is set up where all the connectors inside the Workers directory and the running connectors can be seen. The available installed connectors can be accessed by pressing the “NEW” button. After Selecting a particular connector (Here the PAP-VFI connector) the configuration has to be entered.

D2.3 Development of Toolboxes Integration Connectors

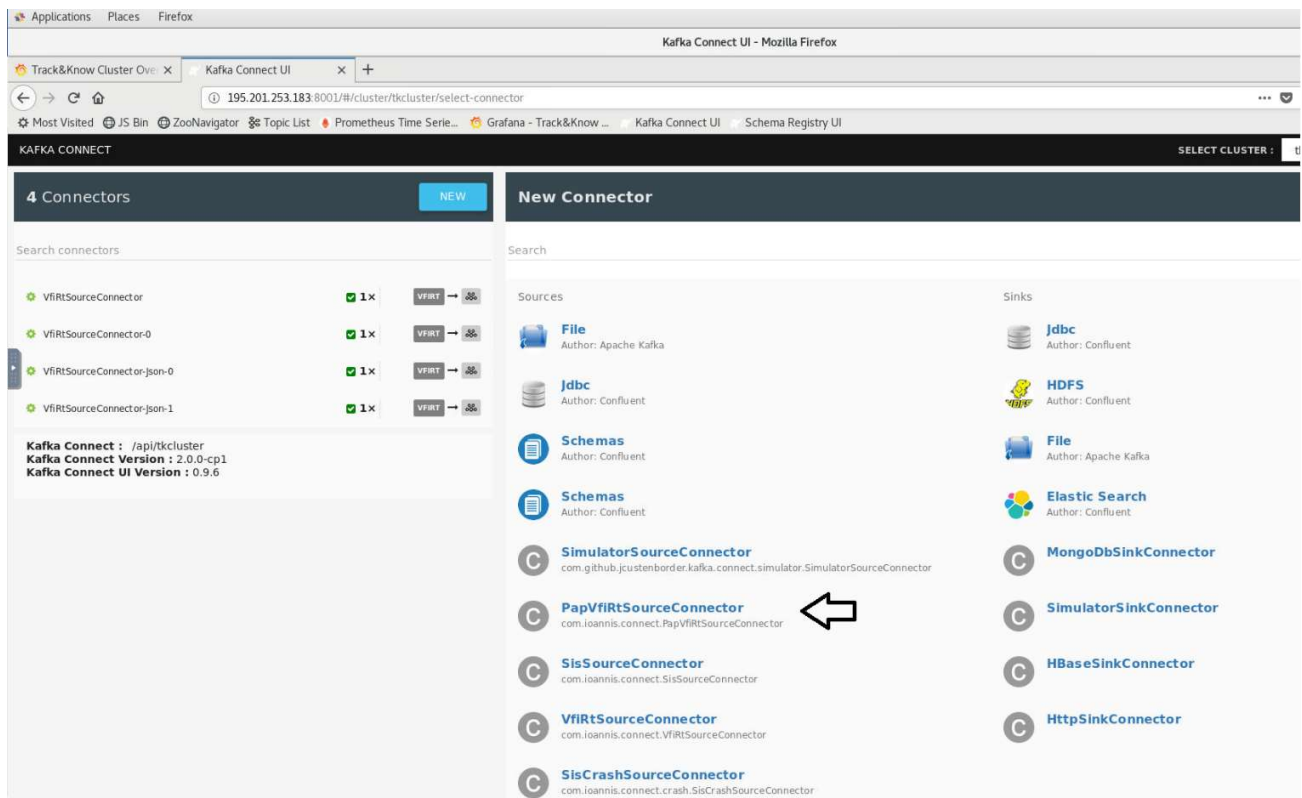


Figure 5.63 Selecting the PAP/VFI connector

In the figure below the PapVfiRtSourceConnector has been selected and its configuration window is shown:

D2.3 Development of Toolboxes Integration Connectors

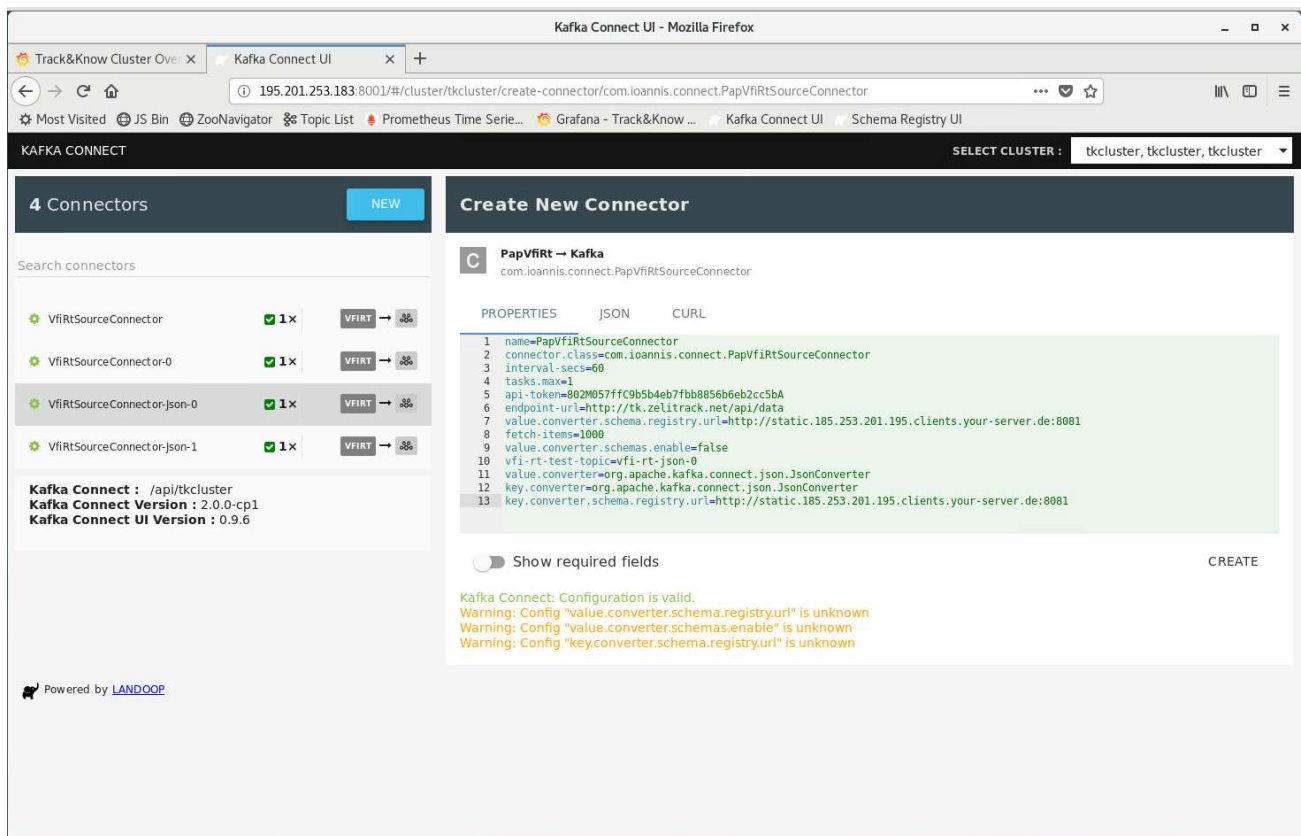


Figure 5.64 Configuring the PAP/VFI connector

In detail, the configuration options are listed in the following table together with their description:

Table 12 Configuration options for the PAP/VFI live data connector

Property and Sample value (Example)	Description
name=PapVfiRtSourceConnector	The name of this connector instance. There may be many instances with different names
connector.class=com.ioannis.connect.PapVfiRtSourceConnector	The type of connector which is actually the connector's java class
interval-secs=60	How often data should be retrieved from the VFI API. At this stage the minimum should be 60 seconds.
tasks.max=1	The number of tasks this connector should start.
api-token=sampleToken7fbb8856b6eb	The API token used – Provided by VFI.
endpoint-url=http://tk.zelitrack.net/api/papdata	The URL of the REST endpoint that provides the data.

value.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The schema registry url where the value schema is held.
fetch-items=100	The number of items (vehicles) for which data will be returned. The maximum allowed is 1000
vfi-rt-topic=d23-pap-vfi-live	The topic where the retrieved data will be written in Kafka
value.converter=org.apache.kafka.connect.json.JsonConverter	The Converter used for the value. Here is JSON but can also be AVRO
key.converter=org.apache.kafka.connect.json.JsonConverter	The Converter used for the Key. Here is JSON but can also be AVRO
key.converter.schema.registry.url=http://static.185.253.201.195.clients.your-server.de:8081	The schema registry url where the key schema is held.

Currently the Kafka Connect PAP/VFI Connector has been implemented and tested and it is ready to begin introducing data originating from the usage of the VFI developed smartphone logger app. The connector will operate in a similar fashion to the VFI live fleet data connector presented in section 5.1.2 and in an “always on” mode. The connector is capable of accumulating vehicle data in JSON and Avro formats with a single and multiple partitions in the target topics. Toolboxes can access that data from their beginning or any other offset and at whatever rate is applicable. Of course, a client can connect to the live topic and begin receiving new data from that point onwards.

Due to the incurring costs of transferring data over mobile networks the VFI/PAP logger app provides the functionality of storing the data and uploading at a later stage, once WIFI is available. This approach although it does not provide live data, it is probable to be employed in planned experiments as it will not interfere with the participants mobile network traffic allowances and charging. For the purpose of introducing the data that has been accumulated by VFI through the app operating in this mode, the connector used for the historical fleet data and presented in section 5.1.1 of this document can be used.

5.4 Other available Connectors

Apart from the presented custom connectors which have been developed specifically for the Track&Know data sources, it is worth noting some other approaches that are available to be used if there is a need.

5.4.1 Introducing data by using the Rest Proxy

The Confluent REST Proxy is part of Confluent Open Source distribution and provides the functionality of a well-defined REST interface to an Apache Kafka deployment. The interface provides the ability to list, create and delete topics, retrieve information with respect to the partitioning of topics and the distribution of the partitions between available brokers, produce or consume messages from a specific topic while also being able to retrieve details about message offsets, and also retrieve information regarding the number of Cluster Brokers. In addition to the functionality mentioned, it is also possible to alter Producer and Consumer configurations.

The importance of the REST Proxy becomes apparent in cases where there is a need to interact with an application implemented using technologies that are not supported already by Kafka or in cases where an administrative interface is built to better manage and review the state of the Cluster. In addition to that it allows open source tools that are dependent on the Rest Proxy to be enabled and used to interact with the cluster, further increasing the offered functionality to users. The Proxy also makes possible the scripting of various administrative actions while enabling a stream processing framework that doesn't support Kafka natively to effectively use it as a data source. Finally, it makes possible to write messages to a specific topic by making REST calls to the API which is the reason it is presented as an additional way to connect data to the Platform.

5.4.2 Connectors available in the Confluent Hub

The Confluent Hub [15] is the dedicated place where existing connectors can be found and can be used for interconnecting a broad set of technologies and systems. Both Sources and Sinks are readily available which are supported at various levels by Confluent or other renowned software companies, complemented by connectors provided by and supported from the Community. Although the list of available connectors is very large, some worth noting include the JDBC, HDFS, Couchbase DB, MongoDB, Elasticsearch, HBASE, Solr and Twitter to name a few, while a full list can be found in [15]. Connectors can be run in both standalone mode or in a distributed mode since a cluster is available in the Track&Know Platform which offers the scalability and automatic load balancing characteristics that enable the connectors to support an entire organization by adding more workers to a Cluster [9].

6 Conclusions

Track & Know aims to develop a new software framework to increase the efficiency of Big Data applications in the transport, mobility, motor insurance and health service sectors. With an efficient, scalable, industry-proven communication platform serving as its “central nervous system” it is aimed to enable solutions that seamlessly integrate truly capable Big Data toolboxes that can cope with increased loads of information irrespective to the domain in question.

The Platform integrates and introduces the data sources made available by the Track&Know consortium by extensively employing the modern and performant connector technologies that have been presented. By introducing the data sources it supports and facilitates further the development of modular, generic, reusable user-friendly toolboxes that will be readily applicable in the addressed markets. The Track&Know Platform will provide the opportunity to all Pilot and Toolbox partners to utilise cutting edge and efficient approaches to overcome barriers while gaining experience in how to best utilise new technologies and components on-boarded by the Platform. The execution of the Pilots will demonstrate how these new technologies have indeed improved their processes and products leading to cost, time and green efficiencies.

The proposed platform provides a robust, scalable and re-deployable infrastructure, addressing some of the challenges and limitations faced within the Big Data industry and provides a framework to utilise and engage with real-world datasets and problems by the technical partners during the course of development and optimisation of toolboxes and the execution of Pilots.

7 References

- [1] Track&Know, "D6.1 Experiments Planning and Setup," June 2018.
- [2] Track&Know, "D2.1 Architectures for the management of structured & unstructured data streams," December 2018.
- [3] DataFlair, "Apache Kafka Use cases | Kafka Applications," DataFlair, [Online]. Available: <https://data-flair.training/blogs/kafka-use-cases-applications/>. [Accessed 26 04 2018].
- [4] A. W. Clemans Vasters, "Why does Kafka scale better than other messaging systems like RabbitMQ?," Quora, [Online]. Available: <https://www.quora.com/Why-does-Kafka-scale-better-than-other-messaging-systems-like-RabbitMQ>. [Accessed 25 04 2018].
- [5] N. Narkhede, "Exactly-once Semantics are Possible: Here's How Kafka Does it," Confluent Inc, 30 06 2017. [Online]. Available: <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>. [Accessed 05 04 2018].
- [6] T. A. S. Foundation, "Apache Kafka – Documentation," The Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/documentation/>. [Accessed 26 04 2018].
- [7] Apache, "Apache Kafka - Introduction," The Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/intro>. [Accessed 05 04 2018].
- [8] J. Kreps, "It's Okay To Store Data In Apache Kafka," Confluent Inc, [Online]. Available: <https://www.confluent.io/blog/okay-store-data-apache-kafka/>. [Accessed 05 04 2018].
- [9] Confluent, "Kafka Connect - Confluent Platform," Confluent Inc, [Online]. Available: <https://docs.confluent.io/3.0.0/connect/intro.html>. [Accessed 05 04 2018].
- [10] Confluent, "Connectors and Apache Kafka Connect APIs," Confluent Inc, [Online]. Available: <https://www.confluent.io/connectors/>. [Accessed 13 11 2018].
- [11] Hetzner, "Truly thrifty cloud hosting," Hetzner Online GmbH, [Online]. Available: <https://www.hetzner.com/cloud>. [Accessed 16 11 2018].
- [12] noVNC, "noVNC Open Source VNC Client," noVNC, [Online]. Available: <https://novnc.com/info.html>. [Accessed 16 11 2018].
- [13] Yahoo, "Kafka Manager," Yahoo, [Online]. Available: <https://github.com/yahoo/kafka-manager>. [Accessed 10 10 2018].
- [14] Landoop, "Kafka Connect UI," Landoop, [Online]. Available: <https://github.com/Landoop/kafka-connect-ui>. [Accessed 10 10 2018].

- [15] Confluent, "Confluent Hub," Confluent Inc, [Online]. Available: <https://www.confluent.io/hub/>. [Accessed 05 04 2018].
- [16] I. Pivotal Software, "Understanding When to use RabbitMQ or Apache Kafka," [Online]. Available: <https://content.pivotal.io/rabbitmq/understanding-when-to-use-rabbitmq-or-apache-kafka>. [Accessed 06 11 2018].
- [17] Oracle, "Using JConsole," Oracle Corporation, [Online]. Available: <https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>. [Accessed 26 04 2018].
- [18] A. Duprat, "Choosing a Message Queue," Medium, [Online]. Available: <https://medium.com/linagora-engineering/how-to-choose-a-message-queue-247dde46e66c>. [Accessed 01 04 2018].
- [19] Y. Trudeau, "Exploring Message Brokers: RabbitMQ, Kafka, ActiveMQ, and Kestrel," DZone, [Online]. Available: <https://dzone.com/articles/exploring-message-brokers>. [Accessed 05 04 2018].
- [20] Pivotal, "RabbitMQ," Pivotal Software, [Online]. Available: <https://www.rabbitmq.com/>. [Accessed 05 04 2018].
- [21] Apache, "Apache Kafka," The Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/>. [Accessed 05 04 2018].
- [22] Apache, "Apache ActiveMQ," The Apache Software Foundation, [Online]. Available: <http://activemq.apache.org/>. [Accessed 02 04 2018].
- [23] Pivotal, "What can RabbitMQ do for you?," Pivotal Software, [Online]. Available: <https://www.rabbitmq.com/features.html>. [Accessed 10 04 2018].
- [24] Apache, "Apache Camel How does Camel work with ActiveMQ," The Apache Software Foundation, [Online]. Available: <http://camel.apache.org/how-does-camel-work-with-activemq.html>. [Accessed 25 04 2018].
- [25] Atomikos, "Review: Understanding Message Brokers - Kafka versus ActiveMQ," Atomikos BVBA, [Online]. Available: <https://www.atomikos.com/Blog/ReviewUnderstandingMessageBrokersKafkaVersusActiveMQ>. [Accessed 25 04 2018].
- [26] DataFlair, "Kafka VS RabbitMQ | Difference between RabbitMQ & Kafka," DataFlair, [Online]. Available: <https://data-flair.training/blogs/kafka-vs-rabbitmq/>. [Accessed 29 05 2018].
- [27] M. Shukla, "Apache Kafka Use cases And Applications," LinkedIn, [Online]. Available: <https://www.linkedin.com/pulse/apache-kafka-use-cases-applications-malini-shukla>. [Accessed 20 06 2018].
- [28] J. Kreps, "Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)," LinkedIn Engineering, [Online]. Available: <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>. [Accessed 05 04 2018].

- [29] K. S. E. Philippe Dobbelaere, "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper," *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, June 19-23, 2017, Barcelona, Spain* .
- [30] J. Rao, "The value of Apache Kafka in Big Data ecosystem," Confluent inc, [Online]. Available: <https://www.confluent.io/blog/the-value-of-apache-kafka-in-big-data-ecosystem/>. [Accessed 05 04 2018].
- [31] D. Gutierrez, "A Brief History of Kafka, LinkedIn's Messaging Platform," Inside BIGDATA, [Online]. Available: <https://insidebigdata.com/2016/04/28/a-brief-history-of-kafka-linkedins-messaging-platform/>. [Accessed 28 04 2018].
- [32] Confluent, "Authorization and ACLs," Confluent Inc., [Online]. Available: <https://docs.confluent.io/3.2.0/kafka/authorization.html>. [Accessed 05 04 2018].
- [33] Apache, "Kafka protocol guide," The Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/protocol.html>. [Accessed 05 04 2018].
- [34] J. Rao, "Clients - Apache Kafka," The Apache Software Foundation, [Online]. Available: <https://cwiki.apache.org/confluence/display/KAFKA/Clients>. [Accessed 08 04 2018].
- [35] Confluent, "Kafka Connect Architecture," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/connect/design.html>. [Accessed 13 11 2018].
- [36] Apache, "Apache Avro 1.8.2 Documentation," The Apache Software Foundation, [Online]. Available: <https://avro.apache.org/docs/current/>. [Accessed 13 11 2018].
- [37] Confluent, "Schema Registry - Confluent Platform," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/schema-registry/docs/index.html>. [Accessed 13 11 2018].
- [38] Google, "Cloud Dataflow - Stream & Batch Data Processing," Google , [Online]. Available: <https://cloud.google.com/dataflow/>. [Accessed 14 11 2018].
- [39] Amazon, "AWS Lambda - Serverless Compute," Amazon Web Services Inc, [Online]. Available: <https://aws.amazon.com/lambda/>. [Accessed 14 11 2018].
- [40] Apache, "Apache Storm," Apache Software Foundation, [Online]. Available: <http://storm.apache.org/>. [Accessed 14 11 2018].
- [41] Apache, "Apache Flink: Stateful Computations over Data Streams," Apache Software Foundation, [Online]. Available: <https://flink.apache.org/>. [Accessed 14 11 2018].
- [42] Apache, "Spark Streaming | Apache Spark," Apache Software Foundation, [Online]. Available: <https://spark.apache.org/streaming/>. [Accessed 14 11 2018].
- [43] Apache, "Storm Kafka Integration," Apache Software Foundation, [Online]. Available: <http://storm.apache.org/releases/2.0.0-SNAPSHOT/storm-kafka.html>. [Accessed 14 11 2018].

- [44] R. Metzger, "Kafka + Flink: A Practical, How-To Guide," dataArtisans, 29 2015. [Online]. Available: <https://data-artisans.com/blog/kafka-flink-a-practical-how-to>. [Accessed 14 11 2018].
- [45] Apache, "Samza," Apache Software Foundation, [Online]. Available: <http://samza.apache.org/>. [Accessed 14 11 2018].
- [46] C. Riccomini, "How LinkedIn Uses Apache Samza," InfoQueue, 09 02 2014. [Online]. Available: <https://www.infoq.com/articles/linkedin-samza>. [Accessed 14 11 2018].
- [47] Apache, "Spark Streaming + Kafka Integration Guide," Apache Software Foundation, [Online]. Available: <https://spark.apache.org/docs/2.2.0/streaming-kafka-integration.html>. [Accessed 14 11 2018].
- [48] Apache, "Apache Kafka - Kafka Streams," Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/documentation/streams/>. [Accessed 14 11 2018].
- [49] Confluent, "Streams DSL," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/streams/developer-guide/dsl-api.html>. [Accessed 14 11 2018].
- [50] J. Kreps, "Introducing Kafka Streams: Stream Processing Made Simple," Confluent Inc, 10 03 2016. [Online]. Available: <https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/>. [Accessed 14 11 2018].
- [51] M. J. S. Florian Troßbach, "Crossing the Streams – Joins in Apache Kafka," Confluent Inc, 19 12 2017. [Online]. Available: <https://www.confluent.io/blog/crossing-streams-joins-apache-kafka/>. [Accessed 15 11 2018].
- [52] Confluent, "KSQL," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/ksql/docs/index.html>. [Accessed 15 11 2018].
- [53] Track&Know, "D1.1 Track and Know Observatory," June 2018.
- [54] Google, "Google Cloud Platform Free Tier," Google, [Online]. Available: <https://cloud.google.com/free/>. [Accessed 16 11 2018].
- [55] D. Stancevic, "Zero Copy I: User-Mode Perspective," linuxjournal, [Online]. Available: <https://www.linuxjournal.com/article/6345?page=0,0>. [Accessed 12 11 2018].
- [56] Elkosmon, "ZooNavigator," [Online]. Available: <https://github.com/elkozmon/zoonavigator>. [Accessed 10 10 2018].
- [57] LinkedIn, "Kafka Monitor," LinkedIn, [Online]. Available: <https://github.com/linkedin/kafka-monitor>. [Accessed 10 10 2018].
- [58] Landoop, "Schema Registry UI," Landoop, [Online]. Available: <https://github.com/Landoop/schema-registry-ui>. [Accessed 10 10 2018].

8 Annex



Big Data for Mobility Tracking Knowledge Extraction in Urban Areas

D2.3 Development of Toolboxes Integration Connectors - ANNEX

Document Summary Information

Grant Agreement No	780754	Acronym	TRACK & KNOW
Full Title	Big Data for Mobility Tracking Knowledge Extraction in Urban Areas		
Start Date	01/01/2018	Duration	36 months
Project URL	https://trackandknow.eu		
Deliverable	D2.3 Development of Toolboxes Integration Connectors ANNEX		
Work Package	WP2 Management Report Annex (BDMI Toolbox)		
Contractual due date	02/08/19	Actual submission date	02/08/19
Nature	Other	Dissemination Level	PU
Lead Beneficiary	05 - INTRASOFT		
Responsible Author	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)		
Contributions from	Toni Staykova (CEL), Leonardo Longhi (SIS), Fabio Manichetti (CNR), Mirco Nanni (CNR), Gennady Andrienko (Fraunhofer), Ian Smith (PAP), Akrivi Vlachou (UPRC), Christos Doulkeridis (UPRC), Yannis Theodoridis (UPRC), Athanasios Koumparos (VFI), Anagnostis Delkos (VFI), Panos Livanos (VFI)		

Revision history (including peer reviewing & quality control)

Version	Issue Date	% Complete	Changes	Contributor(s)
V0.1	15/07/19	5%	Initial Deliverable Structure	Ioannis Daskalopoulos (INTRA)
V0.2	23/07/19	95%	Internal Review version	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)
V0.3	29/07/19	99%	Peer Review Contributions	Ibad Kureshi (INLECOM), Yannis Theodoridis (UPRC)
V0.4	31/08/19	100%	QA process	Marios Logothetis (INTRA)
V1.0	01/08/19	100%	Final version	Ioannis Daskalopoulos (INTRA), Marios Logothetis (INTRA)

Disclaimer

The content of the publication herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the TRACK&KNOW consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the TRACK&KNOW Consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the TRACK&KNOW Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© TRACK&KNOW Consortium, 2018-2020. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Executive Summary

This document presents additional information to deliverable “D2.3 Development of Toolboxes Integration Connectors” aiming to further clarify specific components of the Track&Know platform, according to recommendations and feedback received at the midterm project review. In particular, the focus resides on highlighting the internal workings of platform blocks and connectors providing additional documentation on their components.

Table of Contents

Document Summary Information.....	92
Revision history (including peer reviewing & quality control).....	93
Disclaimer	93
Copyright message.....	93
Executive Summary	94
List of Figures.....	96
Glossary of terms and abbreviations used.....	97
1 Introduction.....	100
1.1 Overview and Annex Structure	100
2 Track & Know Architecture and System Design	101
2.1 High-level Architecture	101
2.2 System Architecture	102
3 Track & Know Datasets and Connectors	106
3.1 VFI Data Connectors.....	107
3.1.1 Kafka Producer type Connector for the VFI historical data	107
3.1.2 Kafka Connect type Connector for the VFI live data.....	109
3.2 SIS Data Connectors	110
3.2.1 Kafka Connect type Connector for the SIS METRICS data.....	110
3.2.2 Kafka Connect type Connector for the SIS VOUCHER data.....	111
3.2.3 Kafka Producer type Connector for the SIS METRICS data	113
3.2.4 Kafka Producer type Connector for the SIS VOUCHER data.....	114
3.3 PAP Data Connectors.....	115
3.3.1 Producer type Connector for the PAP reconstructed journey data	115
3.3.2 Kafka Connect type Connector for the VFI/PAP Smartphone app live data	116
4 Conclusions.....	118
5 References	119
6 APPENDIX A to ANNEX	121
6.1 Content Types	121
6.2 Errors	122
6.3 Topics.....	122
6.4 Partitions.....	128
6.5 Consumers	134
6.6 Brokers.....	149
6.7 Topics.....	149
6.8 Partitions.....	155
6.9 Consumers	164
6.10 Brokers.....	170
7 APPENDIX B to ANNEX	171
7.1 Compatibility.....	171
7.2 Content Types	171
7.3 Errors	172
7.4 Schemas.....	172
7.5 Subjects.....	173
7.6 Compatibility.....	180
7.7 Config.....	182
8 APPENDIX C to ANNEX	186
8.1 Content Types	186
8.2 Statuses & Errors.....	186

8.3	Connectors	187
8.4	Tasks	193
8.5	Connector Plugins	195
9	APPENDIX D to ANNEX	199
9.1	Kafka Producer type Connector for the VFI historical data Class Diagram.....	199
9.2	Kafka Connect type Connector for the VFI live data Class Diagram.....	200
9.3	Kafka Connect type Connector for the SIS METRICS data Class Diagram.....	201
9.4	Kafka Connect type Connector for the SIS VOUCHER data Class Diagram	202
9.5	Kafka Producer type Connector for the SIS METRICS data Class Diagram	203
9.6	Kafka Producer type Connector for the SIS VOUCHER data Class Diagram.....	204
9.7	Kafka Producer type Connector for the PAP reconstructed journey data Class Diagram	205
9.8	Kafka Connect type Connector for the VFI/PAP Smartphone app live data Class Diagram	206

List of Figures

Figure 2.1	Track&Know High level architecture	101
Figure 2.2	Track&Know Platform Cloud Computing Nodes	102
Figure 3.1	The Track&Know repository	107
Figure 3.2	Kafka Producer type Connector for the VFI historical data Class Diagram.....	108
Figure 3.3	Kafka Connect type Connector for the VFI live data Class Diagram.....	109
Figure 3.4	Kafka Connect type Connector for the SIS METRICS data Class Diagram.....	110
Figure 3.5	Kafka Connect type Connector for the SIS VOUCHER data Class Diagram	112
Figure 3.6	Kafka Producer type Connector for the SIS METRICS data Class Diagram	113
Figure 3.7	Kafka Producer type Connector for the SIS VOUCHER data Class Diagram.....	114
Figure 3.8	Producer type Connector for the PAP reconstructed journey data Class Diagram	115
Figure 3.9	Kafka Connect type Connector for the VFI/PAP Smartphone app live data Class Diagram	116
Figure 9.1	Kafka Producer type Connector for the VFI historical data Class Diagram.....	199
Figure 9.2	Kafka Connect type Connector for the VFI live data Class Diagram.....	200
Figure 9.3	Kafka Connect type Connector for the SIS METRICS data Class Diagram.....	201
Figure 9.4	Kafka Connect type Connector for the SIS VOUCHER data Class Diagram	202
Figure 9.5	Kafka Producer type Connector for the SIS METRICS data Class Diagram	203
Figure 9.6	Kafka Producer type Connector for the SIS VOUCHER data Class Diagram.....	204
Figure 9.7	Kafka Producer type Connector for the PAP reconstructed journey data Class Diagram	205
Figure 9.8	Kafka Connect type Connector for the VFI/PAP Smartphone app live data Class Diagram	206

Glossary of terms and abbreviations used

Abbreviation / Term	Description
API	Application Programmable Interface
AMQP	Advanced Message Queueing Protocol
BD	Big Data
BDA	Big Data Analytics
BDP	Big Data Processing
BMDI	Big Mobility Data Integrator
BMI	Body Mass Index
CCG	Clinical Commissioning Groups
CER	Complex Event Recognition
CLI	Command Line Interface
CPAP	Continuous Positive Airway Pressure
CPU	Central Processing Unit
CSV	Comma Separated Values
DoA	Description of Action
DB	Database
DNA	Did Not Attend
DST	Day-light Saving Time
DVLA	Driver and Vehicle Licensing Authority
ESS	Epworth Sleepiness Scale
EtC	Ethics Committee
ETL	Extract Transform Load
FTP	File Transfer Protocol
GPS	Global Positioning System
GUI	Graphical User Interface

HBASE	Hadoop Database
HDFS	Hadoop Distributed File System
HGV	Heavy Goods Vehicle
HTTPS	Hypertext Transfer Protocol Secure
IO	Input/output
IT	Information Technology
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extensions
JSON	Java Script Object Notation
JVM	Java Virtual Machine
KPI	Key Performance Indicators
MQTT	Message Queue Telemetry Transport
NFS	Network File System
NIST	National Institute of Standards and Technologies
ODI	Oxygen Desaturation Index
OSA	Obstructive Sleep Apnoea
PMB	Project Management Board
PR	Pulse Rate
RPM	Rotations Per Minute
SASL	Simple Authentication and Security Layer
SFTP	SSH File Transfer Protocol
SLA	Service Level Agreement
SQL	Structured Query Language
SSH	Secure Shell
SSHFS	SSH Filesystem

SSL	Secure Sockets Layer
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
TLS	Transport Layer Security
URL	Universal Resource Locator
UTC	Coordinated Universal Time
UV	Ultraviolet
VA	Visual Analytics
VM	Virtual Machine
WP	Work Package

1 Introduction

1.1 Overview and Annex Structure

This Annex presents further insight to specific Track&Know Platform components and to custom Integration Connectors implemented and described in detail in D2.3 [1]. The Annex is produced according to recommendations and feedback received at the midterm project review in M18 of the project.

This Annex is structured in the following way:

- **Chapter 1:** Overview and Annex Structure (this section), outlining the annex and how it relates to the project as a whole.
- **Chapter 2:** Track&Know Architecture and System Design, where the former is presented for reference purposes and the System Design components are further described. **Please note that this distributed deployment is setup for Track&Know specifically. Open source components are used that are configured for the purposes of the project.**
- **Chapter 3:** Track&Know connectors, providing additional information about the VFI, SIS and PAP data connectors complemented by informative figures related to their operation and internal workings. **Please note that these are all new components developed for Track&Know.**
- **Chapter 4:** Conclusions
- **APPENDIX A:** A listing of the REST Proxy API reference.
- **APPENDIX B:** A listing of the Schema Registry API reference.
- **APPENDIX C:** A listing of the Kafka Connect API reference.
- **APPENDIX D:** Detailed Class Diagrams of the developed Connectors.

2 Track & Know Architecture and System Design

The following section aims to present the high-level architecture of the Track&Know platform by highlighting the Data Sources and Data Store, the Connectors and Communication Platform, the underlying Infrastructure, Toolboxes and Pilots. Furthermore, System architecture components are further described, and their functionality is presented.

2.1 High-level Architecture

In this section the Track&Know High level architecture is presented for reference and completeness purposes of this annex, together with a short description of the individual functional components, their interactions and the related workplan's Tasks. This architecture fulfils Big data requirements, described in D1.2, by also considering the data diversity, volume and availability in terms of extremely large and complex collections, and the detailed use-case scenarios described in WP6 and specifically in D6.1.

The architecture consists of:

- Data sources which represent the structured and unstructured data streams to be made available and be connected to the platform.
- Data store which represent the batch and interactive data sources that will be made available and will be connected to the platform.
- Connectors together with the Communication platform, that connect external Data sources and the Data store and make them available to the platform, Toolboxes and Pilots.
- Underlying Infrastructure providing all the necessary Big data tools.

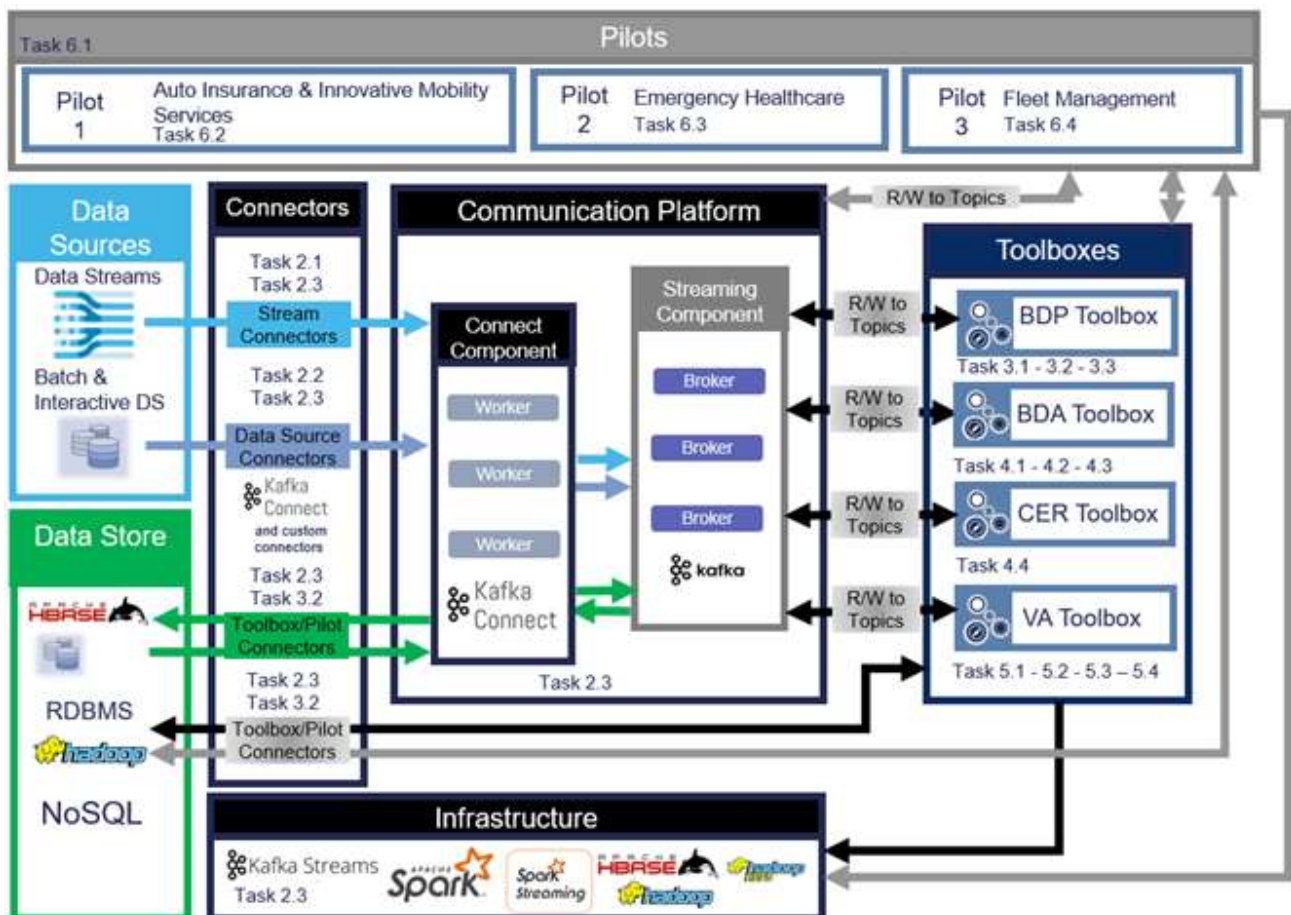


Figure 2.1 Track&Know High level architecture

2.2 System Architecture

The Track&Know Platform is deployed in Hetzner Cloud [2] which provides the necessary flexibility of commissioning and decommissioning virtual machines according to current and future needs in the project. At the time of writing a total of 22 cloud computing instances of varying characteristics are operational, running Centos 7 Linux minimal installations with encrypted disk drives to ensure data encryption at rest. The login configurations for these machines do not permit root login and password authentication, allowing only non-root, key-based authentication via SSH.

Specific firewall rules fully restrict SSH traffic to selected IP addresses. Other traffic is in general only allowed between the cluster members, with the remaining IP addresses blocked by default, isolating the Platform from the outer world. Although the traffic between these hosts is routed internally by Hetzner, which allows for better network performance, the traffic still remains visible to a skilled attacker and therefore should be encrypted. At the time of writing, Hetzner Cloud is not offering a private subnet where the machines can be isolated from the rest of the Internet. In general, all data communications within the cluster are encrypted using SSL (TLS) by employing the encryption mechanisms offered by Apache Kafka. Furthermore, in cases where the data are served to other cluster nodes by an NFS server, then the alternative secure SSHFS is used. Finally, it should be mentioned that any remote desktop connections are performed via HTTPS and file uploads are using SFTP. The diagram below (Figure 4.1) provides an insight to the purpose of each node in the Track&Know Cluster.

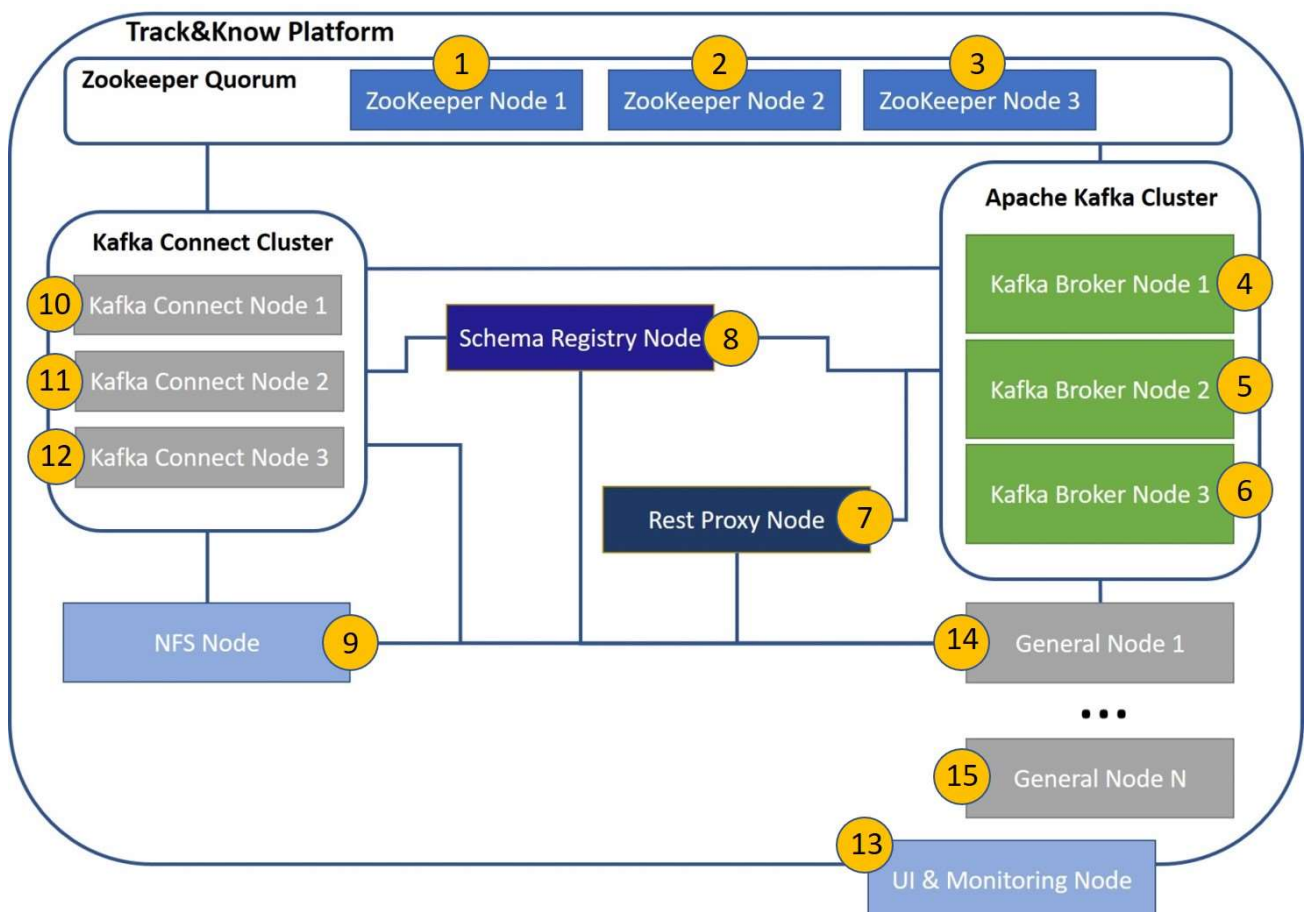


Figure 2.2 Track&Know Platform Cloud Computing Nodes

Please note that this distributed deployment is setup for Track&Know specifically. Open source components are used that are configured for the purposes of the project.

The functionality of each of the above components of the Track&Know platform can be described as follows:

- **Zookeeper Nodes (1, 2, 3) :** The Zookeeper [3] nodes are open source servers which enable highly reliable distributed coordination. The Quorum is used by Kafka Brokers, Kafka Connect, Schema Registry and Rest Proxy nodes and provides a centralised service in which configuration information is maintained and also provides the necessary distributed synchronisation between the components. The Zookeeper nodes are open source components deployed and configured for the Track&Know Platform.
- **Kafka Brokers (4, 5, 6) :** Kafka Brokers [4] receive, persist and make available all the data in the Track&Know Platform. They represent Servers which are capable of receiving, maintaining and making available data in the form of timestamped and with offset messages, that are hosted in topics. Topics are partitioned and spread across all available brokers for load balancing. Clients can subscribe to multiple topics and simultaneously read from them at different partitions and offsets. Similarly, multiple clients can write to one or more topics simultaneously. Partitions can be replicated on more than one brokers for high availability. Apache Kafka Brokers are open source components deployed and configured for the Track&Know Platform.
- **Rest Proxy (7) :** The REST Proxy [5] provides an interface to the Apache Kafka Cluster which among others allows to view its state, interact with the cluster, produce and consume messages, create and delete topics and perform administrative tasks. It is provided for convenience purposes and also enables other administrative tools to work by utilising it. The REST API reference can be found in [5] and is also included in APPENDIX A of this document for convenience. The REST Proxy is an open source component deployed and configured for the Track&Know Platform.
- **Schema Registry (8) :** The Schema Registry [6] provides a means of storing message schemas and a RESTful interface to access and manipulate the former. Message schemas are available for all the clients of the platform making their transmission unnecessary, while at the same time versioning and schema evolution are possible. The Schema Registry is used by Kafka Connect [7] and can also be utilised by custom clients. The REST API reference can be found in [8] and is also included in APPENDIX B of this document for convenience. The Schema Registry is an open source component deployed and configured for the Track&Know Platform.
- **NFS Node (9) :** This node is for the purpose of maintaining large portions of data (e.g. csv files) and making it available over network to the rest of the VMs in the cluster. This is achieved by mounting the volume of this machine over SSHFS. This way other VMs that mount the volume can have simultaneous read access to the data. In a data loading task, connector code running in parallel on several other VMs accesses parts of the data loaded in this node.
- **Kafka Connect Workers (10, 11, 12) :** The Track&Know Platform features 3 Kafka Connect [7] nodes (which can be increased easily and as needs dictate) where Workers are setup in “Distributed mode”. This setup is both horizontally scalable and fault tolerant out of the box. In distributed mode the nodes coordinate and schedule execution of connectors and tasks across all available workers. The Kafka Connect API reference can be found in [9] and is also included in APPENDIX C of this document for convenience. Apache Kafka Connect Workers are open source components deployed and configured for the Track&Know Platform. The resources of these VMs are also utilised in the cases where a Producer type of connector needs to be run on multiple VMs.

- **UI & Monitoring Node (13)** : The central monitoring approach adopted for the Track&Know platform utilises Prometheus [10] JMX exporters which are running on individual cluster nodes and expose a wide range of metrics depending on the node type. All these metrics are gathered at the UI & Monitoring Node where the Prometheus time series collection and processing server resides. All gathered data are available to compose informative dashboards which provide metrics, graphs and panels about the overall Platform status, performance and health. The visualisation of the gathered metrics is achieved by a custom Track&Know Cluster Overview Dashboard. Grafana [11] is used on the UI & Monitoring Node for the creation of custom panels for the Track&Know platform operation, demo and administration purposes and also for WP3, WP4 and WP5 users to be used as templates when developing WP specific dashboards. The current configuration allows the user to get a thorough overview of the Platform status with a single glance at the available metrics and graphs with the added ability to investigate historical performance data. Currently Zookeeper, Apache Kafka, Schema Registry, Rest Proxy and Kafka Connect are monitored. Furthermore, a selection of open source tools for overview and administration tasks concerning the platform including Kafka Manager [12], Kafka Monitor [13], Zoo Navigator [14], Kafka Connect UI [15], Schema Registry UI [16] are also available. The tools mentioned are open source components that are setup and configured to work with the Track&Know Platform. The Grafana Panels are new components implemented specifically for Track&Know.
- **General Nodes (14, 15)** : The General Nodes appearing in Figure 4.1 represent computing nodes that are hosting a range of components of the Track&Know Project from simple Kafka Producers and Consumers, to Kafka Streams applications, Toolboxes' code and necessary technologies to support it etc. These nodes are provided so that the solutions developed by other Toolbox WPs can be accommodated. It should be noted that these nodes are commissioned when a specific need arises, in order for Toolbox and supporting libraries to be configured.

The above components have been selected, installed, configured and fine-tuned to serve the needs of the Project and **it should be noted that the platform they resemble was created specifically for Track&Know**. Although there may exist software tools that can deploy Apache Kafka (mainly on a single host which is not an option for any of the aims of the Project), the distributed deployment presented above, its configuration which makes it work in a distributed, horizontally scalable, secure, fault tolerant fashion and the installation of the various open source tools that complete the platform are not available. The listing below briefly summarises the effort required for the realisation of the above:

- Cloud VM instances creation in Hetzner Cloud [2] according to needs and with various CPU and RAM characteristics. Total of 22 VMs. Commissioning and decommissioning of nodes depending on usage to preserve budget.
- Centos 7 Linux installation on machines mainly minimal installs with desktop functionality whenever necessary. Configured with encrypted volumes. Setup of log rotation, maintenance and updates.
- Generation of necessary user accounts and key-only, non-root SSH remote login configuration.
- Fixed IP address assignment, firewall rules (iptables) configuration and maintenance according to needs, allowing traffic only between Track&Know nodes and selected machines outside the platform.
- Installation and configuration of 3 Apache Kafka Brokers [17] to work as a cluster. Generation of necessary Certificate Authority (CA), truststores and keys for TLS encryption, authentication and authorisation.
- Installation and configuration of 3 Apache Kafka Connect Worker [7] nodes with encrypted communications between them and Kafka Brokers. Generation of necessary keys for the encryption authentication and authorisation.
- Setup of a 3-node Zookeeper Quorum [18] on dedicated machines.

- Configuration of an NFS-SSHFS node to function as a remote volume for other machines that can be mount over the network using SSHFS. This effectively makes visible e.g. csv data loaded on that machine available to other nodes over the network. Setup of mount instructions on several other nodes to mount the remote volume.
- Installation and configuration of the Schema Registry [6] node to work with the Kafka Brokers and Kafka Connect Workers mentioned above. Generation of necessary keys for the encryption authentication and authorisation within the cluster.
- Generation of necessary keys for the encryption authentication and authorisation of WP3, WP4 and WP5 clients against the cluster.
- Installation and configuration of the Rest Proxy [5] node to work with the Kafka Brokers and Kafka Connect Workers. Generation of necessary keys for the encryption authentication and authorisation within the cluster. Configuration to allow only https calls.
- Setup and configuration of TigerVNC [19] and noVNC [20] for remote desktop access. Setup to work explicitly over HTTPS.
- Setup and configuration of Prometheus [10] JMX exporters on all nodes for the purpose of emitting metrics of the installed components.
- Setup of Prometheus monitoring system and time series database for maintaining the emitted metrics on the UI & Monitoring Node of the cluster.
- Setup and configuration of Grafana [11] on the UI & Monitoring Node of the cluster. Creation of custom panels for the Track&Know platform demo and administration purposes. Creation of custom panels for WP3, WP4 and WP5 users to be used as templates when developing platform specific dashboards.
- Installation and configuration of a selection of open source tools for overview and administration tasks of the platform. Configuration to function over secure connections and with the distributed deployment of Track&Know. Tools include Kafka Manager [12], Kafka Monitor [13], ZooNavigator [14], Kafka Connect UI [15], Schema Registry UI [16].
- Creation and setup of several VMs according to the needs of WP3, WP4, WP5 with necessary user accounts and remote desktop functionality.
- Preparation of workshop session and related information, documentation and manuals for other WP users.
- Setup of necessary private git repositories for maintaining the developed software under version control.
- Implementation of the connectors for all the datasets made available by partners, for further information please consult Chapter 3 of this document.

The above list is not meant to be exhaustive but to only provide a brief summary of tasks undertaken for the realisation of the Track&Know Platform.

3 Track & Know Datasets and Connectors

The available datasets within Track&Know originate mainly from partners VFI, SIS and PAP, detailed information for which can be found in deliverable D6.1 [21]. Partner VFI has provided historical fleet mobility data in CSV files organised in anonymised customer folders and has also provided a live data feed of mobility data for the vehicles that it monitors, via a REST API with the data in JSON format.

Partner SIS has provided access to a MongoDB instance, which was setup on their premises for accessing the data made available to the Consortium. The MongoDB contains a total of 5 collections split in 2 databases. Collection DATASET1 contains the main mobility data with relative GPS coordinates while collections CRASH and EVENTS contain accident and other type of important events respectively. The POSITIONS and VOUCHER collections of the database contain places of interest and insurance vouchers information. All the SIS data are stored as MongoDB documents and are retrieved in JSON format.

Regarding the data that partner PAP has made available, they consist of data related to the patients' journeys from residence to clinic, obtained by reconstructing GPS traces, directions, route timings and a poly-line as a GeoJSON for each individual appointment from existing appointment data. A plan also exists for introducing a purpose-built smartphone mobility data logger app developed by VFI, which provides patient journey information. It is planned for this data to be made available via a REST API in JSON format, in a similar way to the live feed of fleet mobility data described above. In cases where mobile networking costs should be avoided in a planned experiment, the app can delay transmission of accumulated data until WIFI is available. When operating in this mode the gathered data will be provided by using the same approach as for the VFI historical fleet mobility data. For the patient journeys data, a Producer type connector has been developed whereas for the logger app a Kafka Connect type of connector has been developed.

The data provided by the partners are loaded to the Track&Know platform by using a set of custom connectors implemented by using the Producer and Kafka Connect Classes and Libraries which is the way of developing connectors for Apache Kafka. Please note that while the connectors make use of the available libraries, they represent new components developed for Track&Know. Two types of connectors have been implemented depending on the case. Producer type of connectors are run as standalone java applications whereas Kafka Connect type of connectors are implemented and packaged as Kafka Connect modules (plugins) that are to be instantiated and run in a distributed Kafka Connect Cluster. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later) and the source code is under version control residing in private Git projects/repositories hosted in GitLab.com:

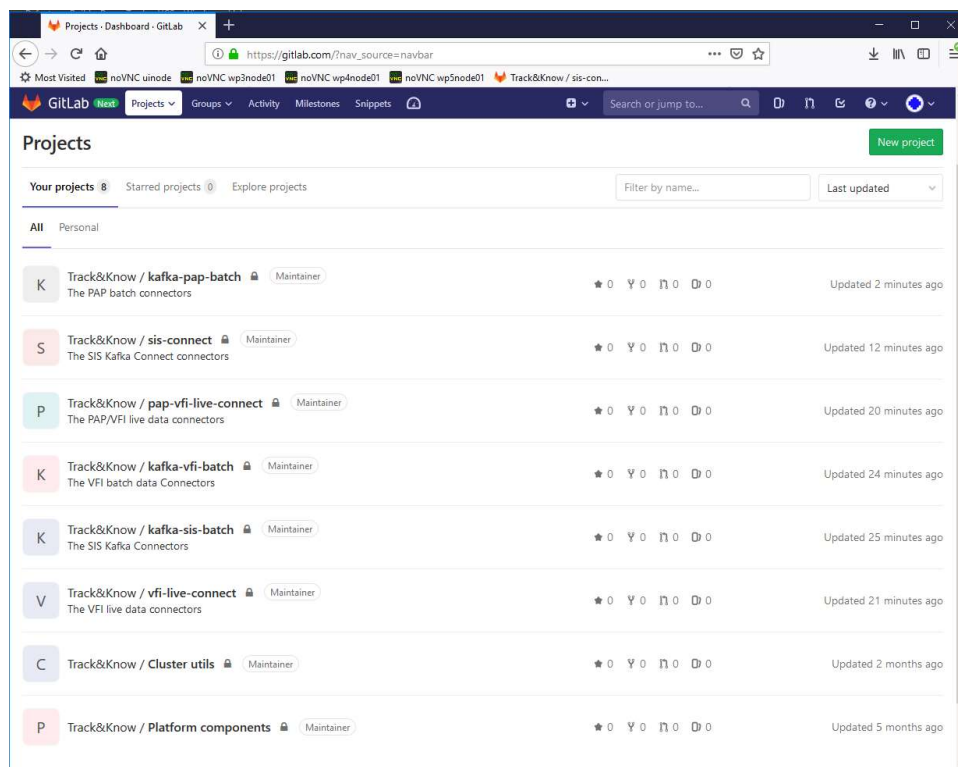


Figure 3.1 The Track&Know repository

The Connectors assume Apache Kafka V2.0 with Kafka Connect in a distributed deployment. In the Track&Know platform a 3-node Apache Kafka Brokers deployment is in place with an additional 3-node Kafka Connect Worker distributed setup. The deployment and usage of connectors is performed on virtual hosts running CentOS Linux (v7.6). It should be noted that all communications between the Connectors and the Track&Know Platform are encrypted (TLS v1.2) and compression is enabled using the Snappy algorithm.

3.1 VFI Data Connectors

In this section the connectors implemented for introducing the VFI data to the Track&Know platform are discussed.

3.1.1 Kafka Producer type Connector for the VFI historical data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

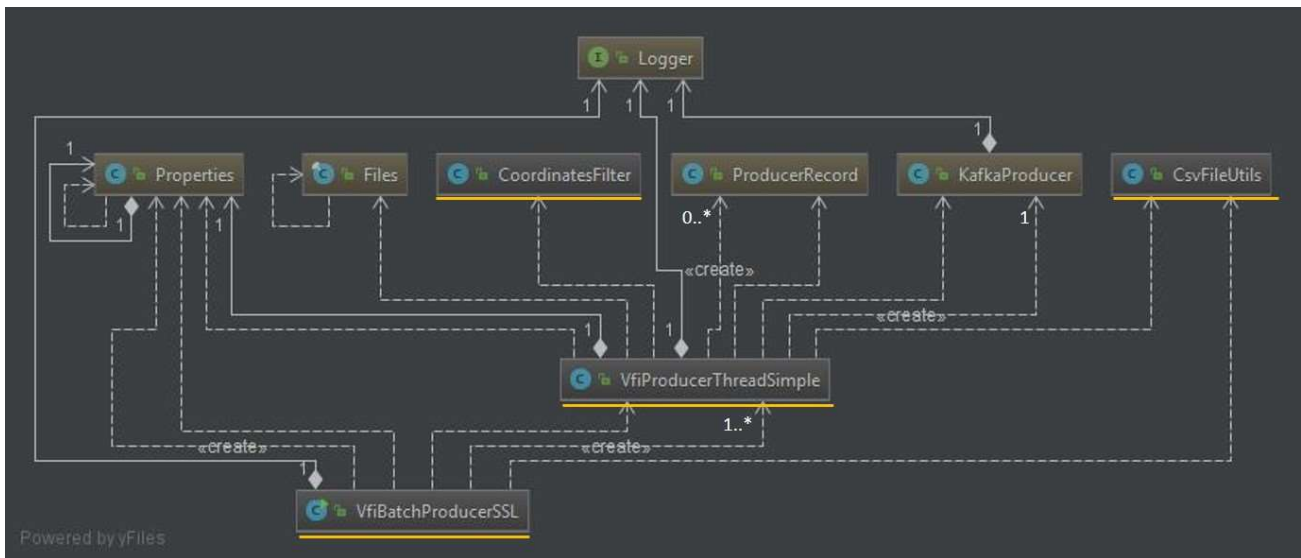


Figure 3.2 Kafka Producer type Connector for the VFI historical data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

When started, the VfiBatchProducerSSL creates a Properties object by reading in the connector configuration file. Then according to the properties, it creates one or multiple VfiProducerThreadSimple instances which are also started. The Properties object is passed to the threads. Each VfiProducerThreadSimple instantiates exactly one KafkaProducer by using the Properties and proceeds to use the Files class in order to access the folders and csv files it is assigned to load from the indicated path in the Properties. For each file the CsvFileUtils may be used if necessary, to remove invalid characters and UTF-8 BOM if it exists. An entire csv file is loaded each time and each one of its lines are used to create from 0 to many Producer Record(s), which are sent asynchronously to the Apache Kafka Brokers by using the KafkaProducer. The KafkaProducer is configured according to the Properties to send ProducerRecord(s) to a specific topic and by using a partitioning scheme which selects the partition inside the topic (in which the records will be written) based on clientID. Also, the KafkaProducer is configured to use the Snappy compression algorithm and a specific key for authentication authorisation and encryption. If configured, the CoordinatesFilter class can be used which rejects lines in the csv files containing GPS coordinates outside Europe. Each VfiProducerThreadSimple which finishes processing its assigned folders exits and the processing ends once all threads have finished. If the properties indicate that multiple instances of the VfiBatchProducerSSL will be run in different hosts then the total task is split among them with each instance assigned a portion according to its ID (configured in the properties). The Logger is instantiated by each component that needs to provide log output.

As mentioned above, the multiple threads that can be configured on each Producer instance allow the maximum CPU utilisation of each Producer node, maximising the throughput. The Producer threads are equally assigned between them several customer folders from the subset that is assigned to this Producer instance.

Once the Producer instances are started, the individual threads begin to read customer folders in parallel. The files inside each customer folder are sorted in a chronological order according to their filename and are one by one loaded and parsed. Each line results in a message that is sent to the Kafka Topic of choice. If the destination Kafka Topic has more than one partition configured, then the messages are introduced to the applicable partition based on the customer number.

3.1.2 Kafka Connect type Connector for the VFI live data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

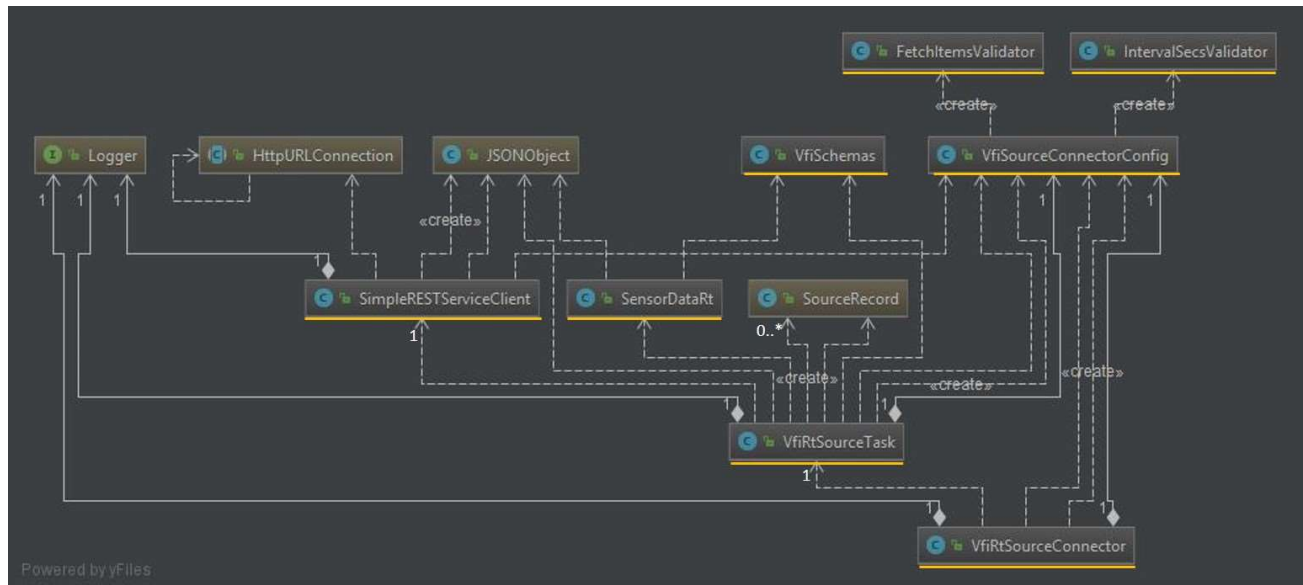


Figure 3.3 Kafka Connect type Connector for the VFI live data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

For the purpose of introducing the VFI live data to the Track&Know Platform the Kafka Connect functionality was used. More specifically a custom Kafka Connect Source Connector was developed which is deployed in a highly available Connect Cluster. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the VFI live data and load them into the topic of choice according to the code of the connector and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the VFI live data will continue to enter the Platform.

The `VfiRtSourceConnector` shown above is packaged as a Kafka Connect module and can be configured and instantiated in the Kafka Connect Cluster. This connector reads the configuration from the `VfiSourceConnectorConfig` which is entered prior to starting the connector and is used to initialise this object. Configuration values that need to be checked in the `VfiSourceConnectorConfig` are checked using the `FetchItemsValidator` and `IntervalSecsValidator` which impose restrictions to the max items to be fetched in each call to the VFI System and the frequency that the data will be fetched at. These validators allow a maximum of 1000 vehicles data every 60 seconds to be fetched according to the (as of now) guidelines from VFI. If the configuration is not valid the connector will not start. Provided that the `VfiRtSourceConnector` manages to validate the configuration, it proceeds to create the necessary `VfiRtSourceTask(s)`. In this implementation the `VfiRtSourceConnector` starts exactly one `VfiRtSourceTask`. While the connector runs, the `VfiRtSourceTask` is assigned to a worker in the Kafka Connect Cluster according to workers load, automatically by Kafka Connect. The `VfiRtSourceTask`, by using values from the `VfiSourceConnectorConfig` proceeds to use a `SimpleRESTServiceClient` which initiates an `HttpURLConnection` to the VFI Server and by making a POST Request it receives a response containing the live vehicle data in JSON format. The response is used to create and initialise a `JSONObject` containing all the received information organised in a `JSONArray`. Each of the 1000 internal `JSONObject` items that the array contains are passed to the constructor of the `SensorDataRt` which creates `SourceRecord` objects according to the `VFISchemas` for the sensor data. The `VFISchemas` class provides the

schema for each Key for the Apache Kafka SourceRecord and the schema for the actual value of the SourceRecord. Depending on the data items retrieved the VfiRtSourceTask creates from 0 to many (max 1000 in this case) SourceRecords that the Kafka Connect Worker running the task will write to the topic selected in the VfiSourceConnectorConfig. The procedure described above occurs repeatedly once the connector is started and at the interval selected (at minimum every 60 seconds). The Logger is instantiated by each component that needs to provide log output.

3.2 SIS Data Connectors

In this section the connectors implemented for introducing the SIS data to the Track&Know platform are discussed. Initially more connectors were implemented (please see Track&Know deliverable D2.3 [1]) as the MongoDB provided by partner SIS included information split between the DATASET1, CRASH, EVENTS and POSITIONS collections which was decided to be merged into METRICS for ease of use within the consortium. Therefore, the METRICS and VOUCHER connectors for the respective collections will be utilized and described but it should be noted that the design and functionality of the other connectors is similar and therefore omitted.

3.2.1 Kafka Connect type Connector for the SIS METRICS data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

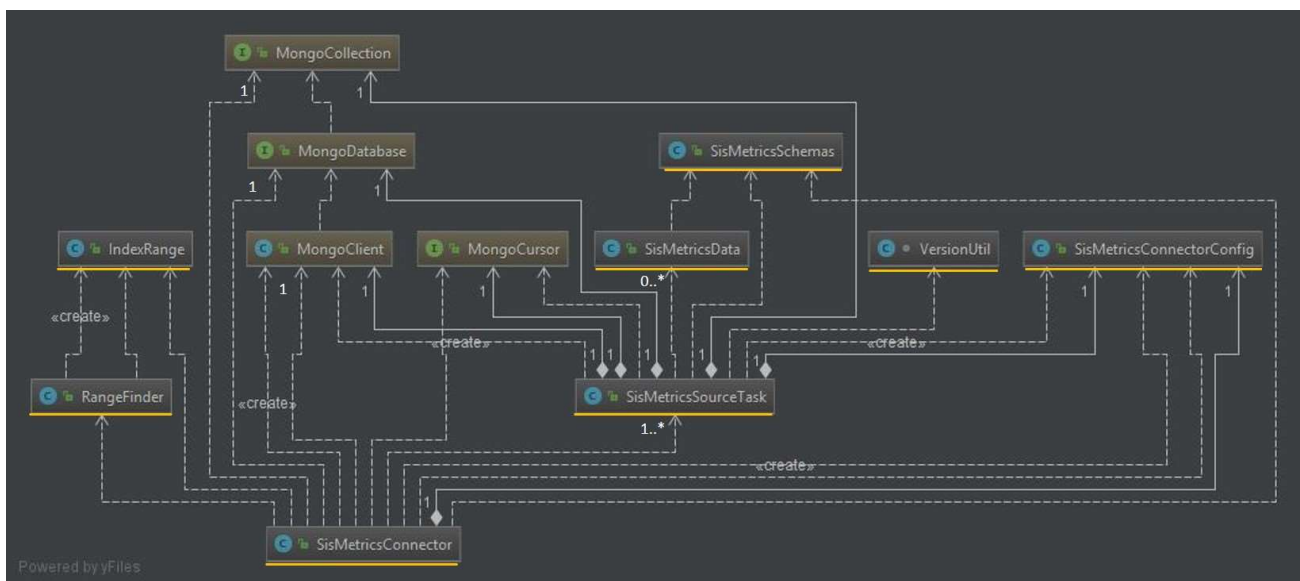


Figure 3.4 Kafka Connect type Connector for the SIS METRICS data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

For the purpose of introducing the SIS METRICS data to the Track&Know Platform the Kafka Connect functionality was used. More specifically a custom Kafka Connect Source Connector was developed which is deployed in a highly available Connect Cluster. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the SIS METRICS data and load it into the topic of choice according to the code of the connector and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the SIS METRICS data will continue to enter the Platform.

The `SisMetricsConnector` shown above is packaged as a Kafka Connect module and can be configured and instantiated in the Kafka Connect Cluster. This connector reads the configuration from the `SisMetricsConnectorConfig` which is entered prior to starting the connector and is used to initialise this object. The `SisMetricsConnector` according to its configuration, it proceeds to create the necessary `SisMetricsSourceTask(s)` which can range from 1 to multiple. In this implementation the `SisMetricsConnector` can start from one to many `SisMetricsSourceTask` instances. In the case that there are multiple instances, the configuration passed to each one is adjusted accordingly to split the effort of loading the data between them. To achieve this, the `SisMetricsConnector` creates and configures a `MongoClient` to connect to the `MongoDatabase` and specifically to the `MongoCollection` of interest (here `METRICS`). By querying the collection, it retrieves the minimum and maximum index which is stored in an `IndexRange` object. Then the initial `IndexRange` object is passed to the `RangeFinder` class which produces the necessary number of index ranges, equal to the number of `SisMetricsSourceTask(s)` that will be run. For example, if the initial range is from 1 to 100 and the `RangeFinder` is asked to provide 2 `IndexRanges` then the output will be 2 ranges from 1 to 50 and 51 to 100 respectively. After preparing the ranges of the collection for which each `SisMetricsSourceTask` will be assigned to work with, the `SisMetricsConnector` can start the tasks. Each task in turn creates a `MongoClient` which connects to the database producing a `MongoDatabase` object as before which is used to get a handle to the collection via a `MongoCollection` object. Each `SisMetricsSourceTask` then proceeds to use its `MongoCollection` object to perform a query to the Collection, requesting data from the range start to the range end values calculated earlier. A `MongoCursor` object is returned once the query is executed which is used to traverse the returned `MongoDB Documents` (records). While there exist more documents available, the `SisMetricsSourceTask` creates `SisMetricsData` objects which result to `SourceRecord` objects according to the `SisMetricsSchemas` for the returned data. The `SisMetricsSchemas` class provides the schema for each Key for an Apache Kafka `SourceRecord` and the schema for the actual value of the `SourceRecord`. The `SourceRecord(s)` are continuously sent to the topic of choice by the individual `SisMetricsSourceTask(s)` which have been started and are run in parallel. It should be noted that each task works with a different index range in the database and therefore with a different portion of the data. Furthermore, each task maintains the last offset of data that it has processed, and it commits that information whenever a message is sent. This way the tasks can be resumed and continue from where they were stopped, making it possible to recover and resume from failure. This is achieved by initialising the tasks at their beginning with that information making it possible to proceed from that point onwards.

3.2.2 Kafka Connect type Connector for the SIS VOUCHER data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

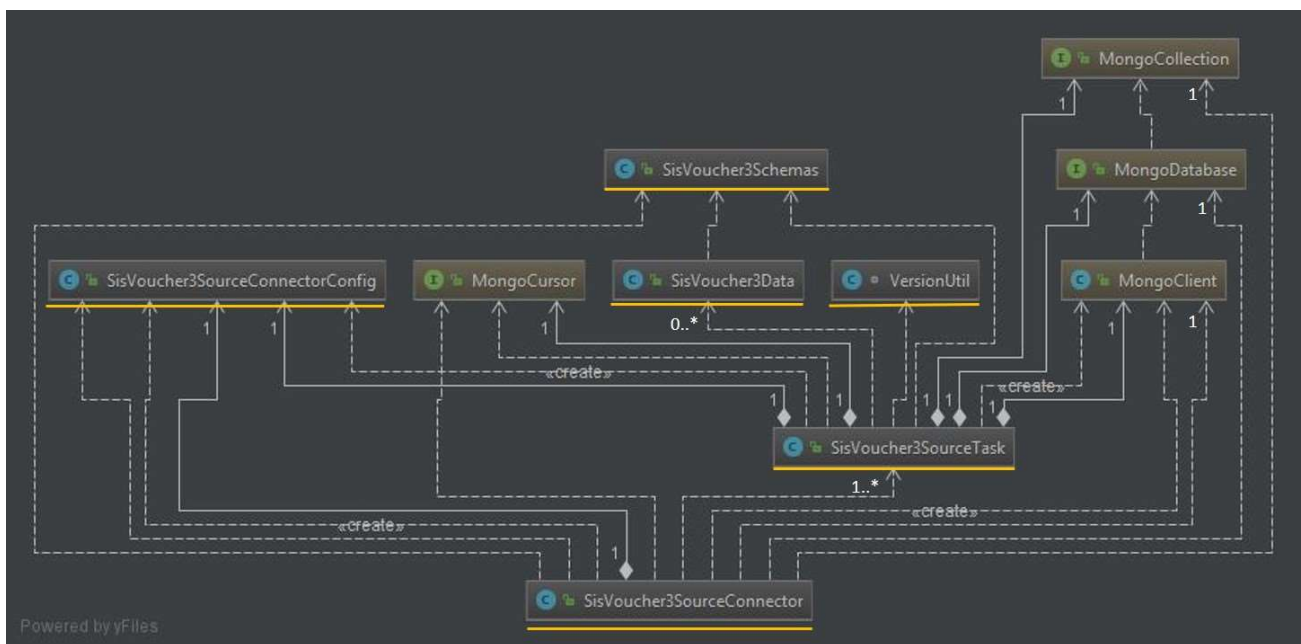


Figure 3.5 Kafka Connect type Connector for the SIS VOUCHER data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

For the purpose of introducing the SIS VOUCHER data to the Track&Know Platform the Kafka Connect functionality was used. More specifically a custom Kafka Connect Source Connector was developed which is deployed in a highly available Connect Cluster. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the SIS VOUCHER data and load it into the topic of choice according to the code of the connector and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the SIS VOUCHER data will continue to enter the Platform.

The `SisVoucher3SourceConnector` shown above is packaged as a Kafka Connect module and can be configured and instantiated in the Kafka Connect Cluster. This connector reads the configuration from the `SisVoucher3SourceConnectorConfig` which is entered prior to starting the connector and is used to initialise this object. The `SisVoucher3SourceConnector` according to its configuration, it proceeds to create the necessary `SisVoucher3SourceTask(s)` which can range from 1 to many depending on the needs. In this implementation the `SisVoucher3SourceConnector` can start from one to many `SisVoucher3SourceTask` instances. In the case that there are multiple instances, the configuration passed to each one is adjusted accordingly to split the effort of loading the data between them. To achieve this, the `SisVoucher3SourceConnector` creates and configures a `MongoClient` to connect to the `MongoDatabase` and specifically to the `MongoCollection` of interest (here `VOUCHER`). By querying the collection, it retrieves the minimum and maximum index which is stored in an `IndexRange` object. Then the initial `IndexRange` object is passed to the `RangeFinder` class which produces the necessary number of index ranges, equal to the number of `SisVoucher3SourceTask(s)` that will be run. For example, if the initial range is from 1 to 100 and the `RangeFinder` is asked to provide 2 `IndexRanges` then the output will be 2 ranges from 1 to 50 and 51 to 100 respectively. After preparing the ranges of the collection for which each `SisVoucher3SourceTask` will be assigned to work with, the `SisVoucher3SourceConnector` can start the tasks. Each task in turn creates a `MongoClient` which connects to the database producing a `MongoDatabase` object as before which is used to get a handle to the collection via a `MongoCollection` object. Each `SisVoucher3SourceTask` then proceeds to use its `MongoCollection` object to perform a query to the Collection, requesting data from the range start to the range end values calculated earlier. A `MongoCursor` object is returned once the query is executed which is used to traverse the returned `MongoDB Documents (records)`. While there

exist more documents available, the `SisVoucher3SourceTask` creates `SisVoucher3Data` objects which result to `SourceRecord` objects according to the `SisVoucher3Schemas` for the returned data. The `SisVoucher3Schemas` class provides the schema for each Key for an Apache Kafka `SourceRecord` and the schema for the actual value of the `SourceRecord`. The `SourceRecord(s)` are continuously sent to the topic of choice by the individual `SisVoucher3SourceTask(s)` which have been started and are run in parallel. It should be noted that each task works with a different index range in the database and therefore with a different portion of the data. Furthermore, each task maintains the last offset of data that it has processed, and it commits that information whenever a message is sent. This way the tasks can be resumed and continue from where they were stopped, making it possible to recover and resume from failure. This is achieved by initialising the tasks at their beginning with that information making it possible to proceed from that point onwards.

3.2.3 Kafka Producer type Connector for the SIS METRICS data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

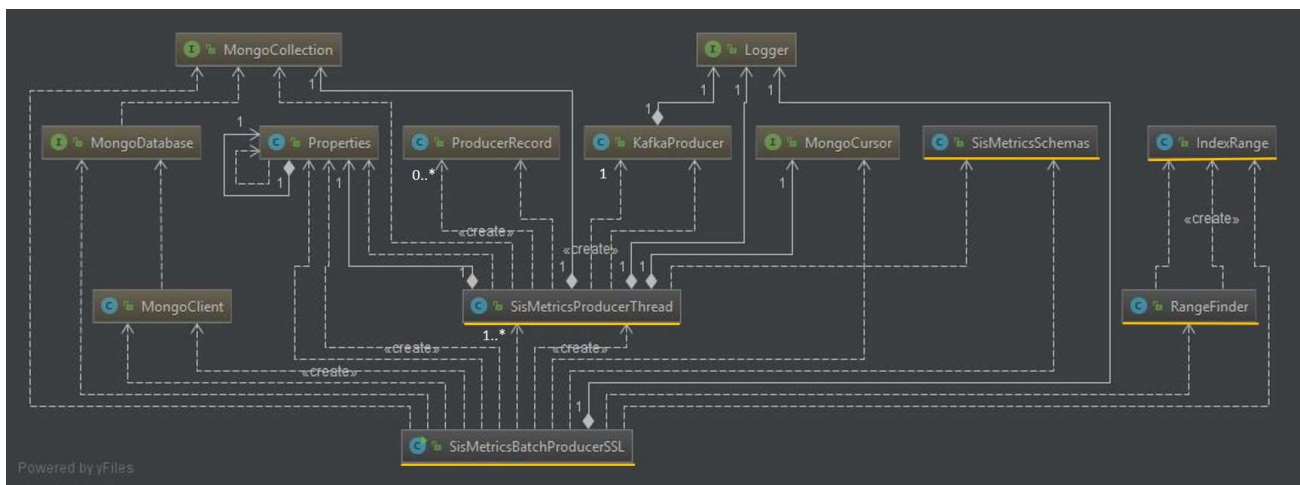


Figure 3.6 Kafka Producer type Connector for the SIS METRICS data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

In addition to the Kafka Connect type of connectors that were presented in the previous sections, a Producer type of connector was also implemented for the SIS METRICS data. This connector was also made available to provide a version that can be run on specific VMs of choice, in the case where a Kafka Connect cluster is not available (e.g. on a VM within partners premises) or if there is a need to have absolute control on the loading task. It should be noted that Kafka Connect type of tasks are assigned and run by the cluster where they may be handed to workers in a non-deterministic fashion and can be restarted without notice when e.g. the cluster rebalances, whereas in a Producer type of Connector the user has full control.

When started, the `SisMetricsBatchProducerSSL` creates a `Properties` object by reading in the connector configuration file. Depending on the configuration, the `SisMetricsBatchProducerSSL` may start from one to many `SisMetricsProducerThread(s)`. Before starting the threads, `SisMetricsBatchProducerSSL` creates and configures a `MongoClient` to connect to the `MongoDatabase` and specifically to the `MongoCollection` of interest (here METRICS) to retrieve the minimum and maximum index which is stored in an `IndexRange` object. Then the initial `IndexRange` object is passed to the `RangeFinder` class which produces the necessary number of index ranges, equal to the number of `SisMetricsProducerThread(s)` that will be started. For example, if the initial range is from 1 to 100 and the `RangeFinder` is asked to provide 2 `IndexRanges` then the output will be 2 ranges from 1 to 50 and 51 to 100 respectively. After preparing the ranges of the collection for which each `SisMetricsProducerThread` will be assigned to process, the `SisMetricsBatchProducerSSL` proceeds to start the `SisMetricsProducerThread(s)`

passing them a `MongoCollection` handle to subsequently perform their queries. A `MongoCursor` object is returned once the query is executed and is used to traverse the returned MongoDB Documents (records). While documents are available, they are used to create from 0 to many `ProducerRecord(s)`, to be sent asynchronously to the Apache Kafka Brokers by using the `KafkaProducer`. The `KafkaProducer` is configured according to the Properties to send `ProducerRecord(s)` to a specific topic and by using a partitioning scheme which selects the partition inside the topic (in which the records will be written) based on vehicleID, retrieved from each document (record) using the schema information contained in `SisMetricsSchemas`. The `KafkaProducer` is configured to use the Snappy compression algorithm and a specific key for authentication, authorisation and encryption. The `Logger` is instantiated by each component that needs to provide log output. Each `SisMetricsProducerThread` which reaches the end of its `MongoCursor` exits and the processing ends once all threads have finished. If the properties indicate that multiple instances of the `SisMetricsBatchProducerSSL` will be run in different hosts then the total task is split among them with each instance assigned a portion according to its ID (configured in the properties).

3.2.4 Kafka Producer type Connector for the SIS VOUCHER data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

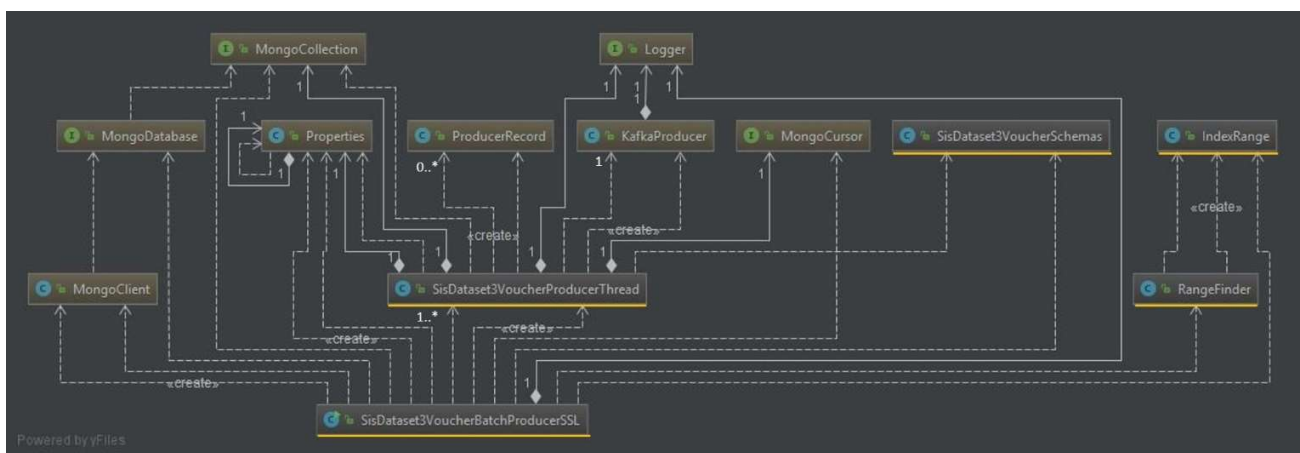


Figure 3.7 Kafka Producer type Connector for the SIS VOUCHER data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

In addition to the Kafka Connect type of connectors that were presented in the previous sections, a Producer type of connector was also implemented for the SIS VOUCHER data. This connector was also made available to provide a version that can be run on specific VMs of choice, in the case where a Kafka Connect cluster is not available (e.g. on a VM within partners premises) or if there is a need to have absolute control on the loading task. It should be noted that Kafka Connect type of tasks are assigned and run by the cluster where they may be handed to workers in a non-deterministic fashion and can be restarted without notice when e.g. the cluster rebalances, whereas in a Producer type of Connector the user has full control.

When started, the `SisDataset3VoucherBatchProducerSSL` creates a `Properties` object by reading in the connector configuration file. Depending on the configuration, the `SisDataset3VoucherBatchProducerSSL` may start from one to many `SisDataset3VoucherProducerThread(s)` according to the configuration. Before starting the threads, `SisDataset3VoucherBatchProducerSSL` creates and configures a `MongoClient` to connect to the `MongoDatabase` and specifically to the `MongoCollection` of interest (here VOUCHER) to retrieve the minimum and maximum index which is stored in an `IndexRange` object. Then the initial `IndexRange` object is passed to the `RangeFinder`

class which produces the necessary number of index ranges, equal to the number of `SisDataset3VoucherProducerThread(s)` that will be started. For example, if the initial range is from 1 to 100 and the `RangeFinder` is asked to provide 2 `IndexRanges` then the output will be 2 ranges from 1 to 50 and 51 to 100 respectively. After preparing the ranges of the collection for which each `SisDataset3VoucherProducerThread` will be assigned to process, the `SisDataset3VoucherBatchProducerSSL` proceeds to start the `SisDataset3VoucherProducerThread(s)`, passing them a `MongoCollection` handle to subsequently perform their queries. A `MongoCursor` object is returned once the query is executed and is used to traverse the returned MongoDB Documents (records). While documents are available, they are used to create from 0 to many `ProducerRecord(s)`, to be sent asynchronously to the Apache Kafka Brokers by using the `KafkaProducer`. The `KafkaProducer` is configured according to the Properties to send `ProducerRecord(s)` to a specific topic and by using a partitioning scheme which selects the partition inside the topic (in which the records will be written) based on `vehicleID`, retrieved from each document (record) using the schema information contained in `SisDataset3VoucherSchemas`. The `KafkaProducer` is configured to use the Snappy compression algorithm and a specific key for authentication, authorisation and encryption. The `Logger` is instantiated by each component that needs to provide log output. Each `SisDataset3VoucherProducerThread` which reaches the end of its `MongoCursor` exits and the processing ends once all threads have finished. If the properties indicate that multiple instances of the `SisDataset3VoucherBatchProducerSSL` will be run in different hosts then the total task is split among them with each instance assigned a portion according to its ID (configured in the properties).

3.3 PAP Data Connectors

In this section the connectors implemented for introducing the SIS data to the Track&Know platform are discussed.

3.3.1 Producer type Connector for the PAP reconstructed journey data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

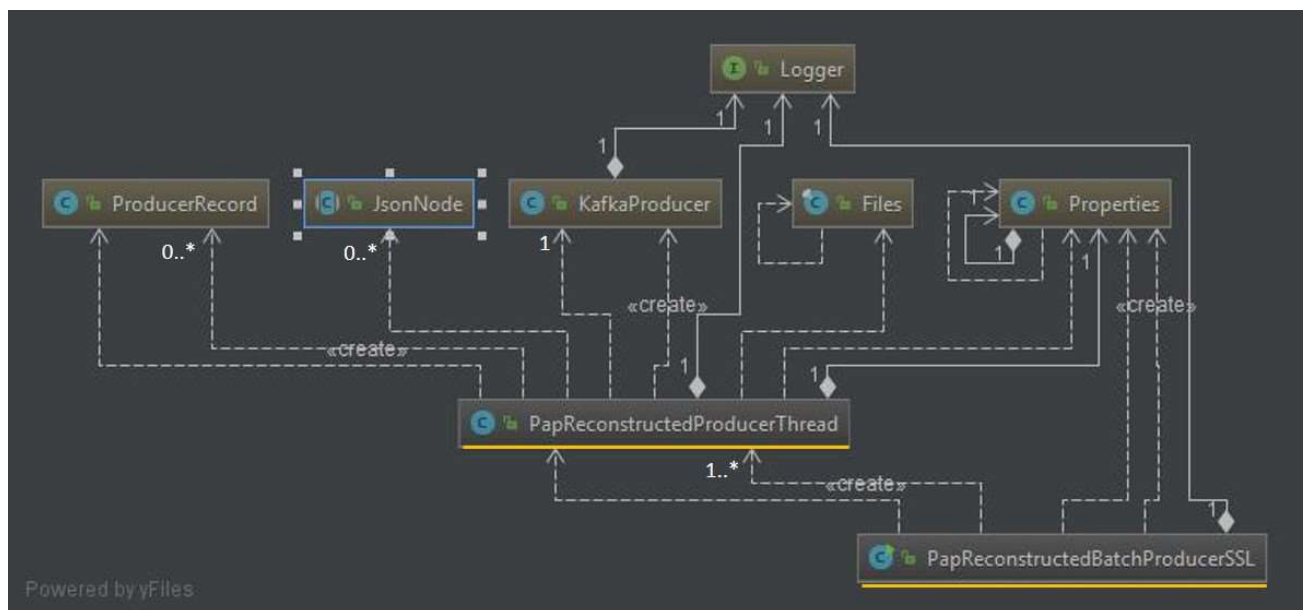


Figure 3.8 Producer type Connector for the PAP reconstructed journey data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

When started, the `PapReconstructedBatchProducerSSL` creates a `Properties` object by reading in the connector configuration file. Then according to the properties, it creates one or multiple `PapReconstructedProducerThread` instances which are also started. The `Properties` object is passed to the threads. Each `PapReconstructedProducerThread` instantiates exactly one `KafkaProducer` by using the `Properties` and proceeds to use the `Files` class in order to access the folders and individual files it is assigned to load from the indicated path in the `Properties`. An entire file is loaded each time, producing a `JsonNode` object. Each one of the `JSON` objects in the `JsonNode` are used to create from 0 to many `ProducerRecord(s)`, to be sent asynchronously to the Apache Kafka Brokers by using the `KafkaProducer`. The `KafkaProducer` is configured according to the `Properties` to send `ProducerRecord(s)` to a specific topic and by using a partitioning scheme which selects the partition inside the topic (in which the records will be written) based on each processed filename. Also, the `KafkaProducer` is configured to use the Snappy compression algorithm and a specific key for authentication, authorisation and encryption. Each `PapReconstructedProducerThread` which finishes processing its assigned folders exits and the processing ends once all threads have finished. If the properties indicate that multiple instances of the `PapReconstructedBatchProducerSSL` will be run in different hosts then the total task is split among them with each instance assigned a portion according to its ID (configured in the properties). The `Logger` is instantiated by each component that needs to provide log output.

As mentioned above, the multiple threads that can be configured on each Producer instance allow the maximum CPU utilisation of each Producer node, maximising the throughput. The Producer threads are equally assigned between them several data folders from the subset that is assigned to this Producer instance.

Once the Producer instances are started, the individual threads begin to read data folders in parallel. The files inside each folder are sorted according to their filename and are one by one loaded as a `JsonNode`. Each object in the `JsonNode` results in a message that is sent to the Kafka Topic of choice. If the destination Kafka Topic has more than one partition configured, then the messages are introduced to the applicable partition based on the filename processed.

3.3.2 Kafka Connect type Connector for the VFI/PAP Smartphone app live data

The figure below presents a simplified class diagram containing only class names to support the discussion. A complete class description with fields and methods can be found in APPENDIX D of this document.

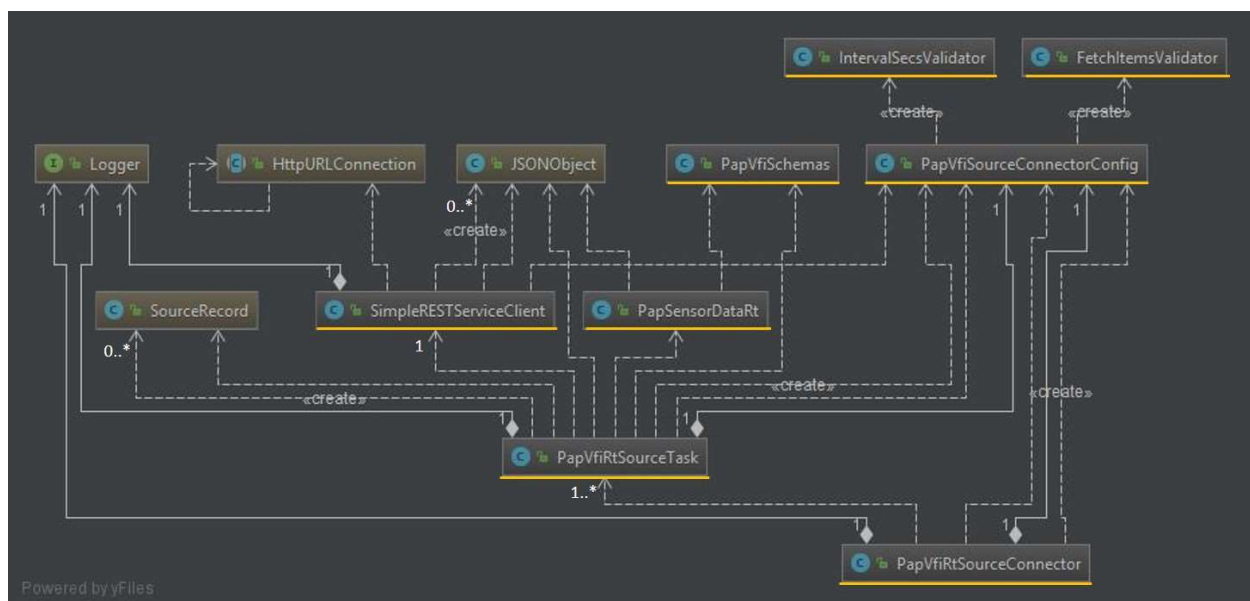


Figure 3.9 Kafka Connect type Connector for the VFI/PAP Smartphone app live data Class Diagram

The classes with the yellow underline represent new code whereas other classes are imported from existing libraries. The language used for the implementation is Java (compatible with JDK 1.8 u31 or later).

For the purpose of introducing the PAP/VFI Smartphone app data to the Track&Know Platform, the Kafka Connect functionality was used. More specifically a custom Kafka Connect Source Connector was developed which is deployed in a highly available Connect Cluster. Once the connector is instantiated the Connect Cluster Workers perform the work of retrieving the PAP/VFI Smartphone app data and load them into the topic of choice according to the code of the connector and its configuration. In the case of failure of a specific worker node, the other remaining nodes of the cluster will continue to run the connector code. This means that even at the case of failure of a node, the PAP/VFI Smartphone app data will continue to enter the Platform.

The PapVfiRtSourceConnector shown above is packaged as a Kafka Connect module and can be configured and instantiated in the Kafka Connect Cluster. This connector reads the configuration from the PapVfiSourceConnectorConfig which is entered prior to starting the connector and is used to initialise this object. Configuration values that need to be checked in the PapVfiSourceConnectorConfig are checked using the FetchItemsValidator and IntervalSecsValidator which impose restrictions to the max items to be fetched in each call to the VFI System and the frequency that the data will be fetched at. These validators allow a maximum of 1000 vehicles data every 60 seconds to be fetched according to the (as of now) guidelines from VFI. If the configuration is not valid the connector will not start. Provided that the PapVfiRtSourceConnector manages to validate the configuration, it proceeds to create the necessary PapVfiRtSourceTask(s). In this implementation the PapVfiRtSourceConnector starts exactly one PapVfiRtSourceTask. While the connector runs, the PapVfiRtSourceTask is assigned to a worker in the Kafka Connect Cluster according to workers load, automatically by Kafka Connect. The PapVfiRtSourceTask, by using values from the PapVfiSourceConnectorConfig proceeds to use a SimpleRESTServiceClient which initiates an HttpURLConnection to the VFI Server and by making a POST Request it receives a response containing the journey data in JSON format. The response is used to create and initialise a JSONObject containing all the received information organised in a JSONArray. Each of the 1000 internal JSONObject items that the array contains are passed to the constructor of the PapSensorDataRt which creates SourceRecord objects according to the PapVfiSchemas for the gathered data. The PapVfiSchemas class provides the schema for each Key for the Apache Kafka SourceRecord and the schema for the actual value of the SourceRecord. Depending on the data items retrieved the VfiRtSourceTask creates from 0 to many (max 1000 in this case) SourceRecords that the Kafka Connect Worker running the task will write to the topic selected in the PapVfiSourceConnectorConfig. The procedure described above occurs indefinitely and at the interval selected (at minimum every 60 seconds). The Logger is instantiated by each component that needs to provide log output.

4 Conclusions

This Annex provided further insight to specific Track&Know Platform components and to custom Integration Connectors implemented and presented in "D2.3 Development of Toolboxes Integration Connectors" [1]. According to recommendations and feedback received at the midterm project review in M18 of the project, additional technical diagrams giving low level information were presented together with API end point information. Furthermore, descriptions of parts developed in the project against ready components were provided and the technologies, programming languages, and other specifications were further highlighted providing further documentation.

5 References

- [1] Track&Know, "D2.3 Development of Toolboxes Integration Connectors," Track&Know, 2019.
- [2] Hetzner, "Truly thrifty cloud hosting," Hetzner Online GmbH, [Online]. Available: <https://www.hetzner.com/cloud>. [Accessed 16 11 2018].
- [3] T. A. S. Foundation, "Welcome to Apache ZooKeeper," The Apache Software Foundation, 2019. [Online]. Available: <https://zookeeper.apache.org/>. [Accessed 18 07 2019].
- [4] J. Laskowski, "Broker Nodes — Kafka Servers · The Internals of Apache Kafka," GitBook, 2019. [Online]. Available: <https://jaceklaskowski.gitbooks.io/apache-kafka/kafka-brokers.html>. [Accessed 18 07 2019].
- [5] Confluent, "REST Proxy," Confluent inc, 2019. [Online]. Available: <https://docs.confluent.io/current/kafka-rest/index.html>. [Accessed 17 07 2019].
- [6] Confluent, "Schema Registry - Confluent Platform," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/schema-registry/docs/index.html>. [Accessed 13 11 2018].
- [7] Confluent, "Kafka Connect Architecture," Confluent Inc, [Online]. Available: <https://docs.confluent.io/current/connect/design.html>. [Accessed 13 11 2018].
- [8] C. inc, "Schema Registry API Reference," Confluent inc, 2019. [Online]. Available: <https://docs.confluent.io/current/schema-registry/develop/api.html>. [Accessed 19 07 2019].
- [9] Confluent, "Kafka Connect REST Interface," Confluent, 2019. [Online]. Available: <https://docs.confluent.io/current/connect/references/restapi.html>. [Accessed 19 07 2019].
- [10] P. Authors, "Prometheus - Monitoring system & time series database," The Linux Foundation, 2019. [Online]. Available: <https://prometheus.io/>. [Accessed 18 07 2019].
- [11] G. Labs, "Grafana - The open platform for analytics and monitoring," Grafana Labs, 2019. [Online]. Available: <https://grafana.com/>. [Accessed 18 07 2019].
- [12] Yahoo, "Kafka Manager," Yahoo, [Online]. Available: <https://github.com/yahoo/kafka-manager>. [Accessed 10 10 2018].
- [13] LinkedIn, "Kafka Monitor," LinkedIn, [Online]. Available: <https://github.com/linkedin/kafka-monitor>. [Accessed 10 10 2018].
- [14] Elkosmon, "ZooNavigator," [Online]. Available: <https://github.com/elkozmon/zoonavigator>. [Accessed 10 10 2018].
- [15] Landoop, "Kafka Connect UI," Landoop, [Online]. Available: <https://github.com/Landoop/kafka-connect-ui>. [Accessed 10 10 2018].

- [16] Landoop, "Schema Registry UI," Landoop, [Online]. Available: <https://github.com/Landoop/schema-registry-ui>. [Accessed 10 10 2018].
- [17] Apache, "Apache Kafka," The Apache Software Foundation, [Online]. Available: <https://kafka.apache.org/>. [Accessed 05 04 2018].
- [18] T. A. S. Foundation, "Apache Zookeeper," The Apache Software Foundation, 2019. [Online]. Available: <https://zookeeper.apache.org/>. [Accessed 17 07 2019].
- [19] TigerVNC, "TigerVNC," TigerVNC, 2019. [Online]. Available: <https://tigervnc.org/>. [Accessed 18 07 2019].
- [20] noVNC, "noVNC Open Source VNC Client," noVNC, [Online]. Available: <https://novnc.com/info.html>. [Accessed 16 11 2018].
- [21] Track&Know, "D6.1 Experiments Planning and Setup," June 2018.
- [22] C. inc, "Confluent REST Proxy API Reference," Confluent, 2019. [Online]. Available: <https://docs.confluent.io/current/kafka-rest/api.html>. [Accessed 17 07 2019].
- [23] C. inc, "Schema Registry API Reference," Confluent, 2019. [Online]. Available: <https://docs.confluent.io/current/schema-registry/develop/api.html>. [Accessed 19 07 2019].

6 APPENDIX A to ANNEX

In this appendix the REST Proxy API Reference is quoted as it appears in the relative online resource [22] for convenience purposes and as requested:

The material in this appendix is Copyrighted by Confluent, Inc (© Copyright 2019, Confluent, Inc).

6.1 Content Types

The REST proxy uses content types for both requests and responses to indicate 3 properties of the data: the serialization format (e.g. `json`), the version of the API (e.g. `v2`), and the *embedded format* (e.g. `json`, `binary` or `avro`). Currently, the only serialization format supported is `json` and the versions of the API are `v1` and `v2`.

The embedded format is the format of data you are producing or consuming, which are embedded into requests or responses in the serialization format. For example, you can provide `binary` data in a `json`-serialized request; in this case the data should be provided as a base64-encoded string and the content type will be `application/vnd.kafka.binary.v2+json`. If your data is just JSON, you can use `json` as the embedded format and embed it directly; in this case the content type will be `application/vnd.kafka.json.v2+json`. The proxy also supports `avro`, in which case a JSON form of the data can be embedded directly and a schema (or schema ID) should be included with the request. If Avro is used, the content type will be `application/vnd.kafka.avro.v2+json`.

The format for the content type is:

```
application/vnd.kafka[.embedded_format].[api_version]+[serialization_format]
```

Copy

The serialization format can be omitted when there are no embedded messages (i.e. for metadata requests you can use `application/vnd.kafka.v2+json`). The preferred content type is `application/vnd.kafka.[embedded_format].v1+json`. However, other less specific content types are permitted, including `application/vnd.kafka+json` to indicate no specific API version requirement (the most recent stable version will be used), `application/json`, and `application/octet-stream`. The latter two are only supported for compatibility and ease of use. In all cases, if the embedded format is omitted, `binary` is assumed. Although using these less specific values is permitted, to remain compatible with future versions you *should* specify preferred content types in requests and check the content types of responses.

Your requests *should* specify the most specific format and version information possible via the HTTP `Accept` header:

```
Accept: application/vnd.kafka.v2+json
```

Copy

The server also supports content negotiation, so you may include multiple, weighted preferences:

```
Accept: application/vnd.kafka.v2+json; q=0.9, application/json; q=0.5
```

Copy

which can be useful when, for example, a new version of the API is preferred but you cannot be certain it is available yet.

6.2 Errors

All API endpoints use a standard error message format for any requests that return an HTTP status indicating an error (any 400 or 500 statuses). For example, a request entity that omits a required field may generate the following response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.kafka.v1+json

{
  "error_code": 422,
  "message": "records may not be empty"
}
```

Copy

Although it is good practice to check the status code, you may safely parse the response of any non-DELETE API calls and check for the presence of an `error_code` field to detect errors.

Some error codes are used frequently across the entire API and you will probably want to have general purpose code to handle these, whereas most other error codes will need to be handled on a per-request basis.

ANY *

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found.
 - Error code 40402 -- Partition not found.
- [422 Unprocessable Entity](#) -- The request payload is either improperly formatted or contains semantic errors
- [500 Internal Server Error](#) --
 - Error code 50001 -- Zookeeper error.
 - Error code 50002 -- Kafka error.
 - Error code 50003 -- Retriable Kafka error. Although the operation failed, it's possible that retrying the request will be successful.
 - Error code 50101 -- Only SSL endpoints were found for the specified broker, but SSL is not supported for the invoked API yet.

6.3 Topics

The topics resource provides information about the topics in your Kafka cluster and their current state. It also lets you produce messages by making `POST` requests to specific topics.

GET /topics

Get a list of Kafka topics.

Response JSON Object:

- **topics** (*array*) -- List of topic names

Example request:

```
GET /topics HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

["topic1", "topic2"]
```

Copy

GET /topics/(*string:topic_name*)

Get metadata about a specific topic.

- Parameters:**
- **topic_name** (*string*) -- Name of the topic to get metadata about

Response JSON Object:

- **name** (*string*) -- Name of the topic
- **configs** (*map*) -- Per-topic configuration overrides
- **partitions** (*array*) -- List of partitions for this topic
- **partitions[i].partition** (*int*) -- the ID of this partition
- **partitions[i].leader** (*int*) -- the broker ID of the leader for this partition
- **partitions[i].replicas** (*array*) -- list of replicas for this partition, including the leader
- **partitions[i].replicas[j].broker** (*array*) -- broker ID of the replica
- **partitions[i].replicas[j].leader** (*boolean*) -- true if this replica is the leader for the partition
- **partitions[i].replicas[j].in_sync** (*boolean*) -- true if this replica is currently in sync with the leader

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found

Example request:

```
GET /topics/test HTTP/1.1
Accept: application/vnd.kafka.v2+json
```

Copy

Example response:

HTTP/1.1 200 OK
 Content-Type: application/vnd.kafka.v2+json

```
{
  "name": "test",
  "configs": {
    "cleanup.policy": "compact"
  },
  "partitions": [
    {
      "partition": 1,
      "leader": 1,
      "replicas": [
        {
          "broker": 1,
          "leader": true,
          "in_sync": true,
        },
        {
          "broker": 2,
          "leader": false,
          "in_sync": true,
        }
      ]
    },
    {
      "partition": 2,
      "leader": 2,
      "replicas": [
        {
          "broker": 1,
          "leader": false,
          "in_sync": true,
        },
        {
          "broker": 2,
          "leader": true,
          "in_sync": true,
        }
      ]
    }
  ]
}
```

Copy

POST /topics/(string:topic_name)

Produce messages to a topic, optionally specifying keys or partitions for the messages. If no partition is provided, one will be chosen based on the hash of the key. If no key is provided, the partition will be chosen for each message in a round-robin fashion.

For the `avro` embedded format, you must provide information about schemas and the REST proxy must be configured with the URL to access Schema Registry (`schema.registry.url`). Schemas may be provided as the full schema encoded as a string, or, after the initial request may be provided as the schema ID returned with the first response.

Parameters:

- **topic_name** (*string*) -- Name of the topic to produce the messages to

Request JSON Object:

- **key_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **key_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **value_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **value_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.

Request JSON Array of Objects:

- **records** -- A list of records to produce to the topic.
- **records[i].key** (*object*) -- The message key, formatted according to the embedded format, or null to omit a key (optional)
- **records[i].value** (*object*) -- The message value, formatted according to the embedded format
- **records[i].partition** (*int*) -- Partition to store the message in (optional)

Response JSON Object:

- **key_schema_id** (*int*) -- The ID for the schema used to produce keys, or null if keys were not used
- **value_schema_id** (*int*) -- The ID for the schema used to produce values.

Response JSON Array of Objects:

- **offsets** (*object*) -- List of partitions and offsets the messages were published to
- **offsets[i].partition** (*int*) -- Partition the message was published to, or null if publishing the message failed
- **offsets[i].offset** (*long*) -- Offset of the message, or null if publishing the message failed
- **offsets[i].error_code** (*long*) --

An error code classifying the reason this operation failed, or null if it succeeded.

- 1 - Non-retriable Kafka exception
- 2 - Retriable Kafka exception; the message might be sent successfully if retried
- **offsets[i].error** (*string*) -- An error message describing why the operation failed, or null if it succeeded

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Request includes keys and uses a format that requires schemas, but does not include the `key_schema` or `key_schema_id` fields
 - Error code 42202 -- Request includes values and uses a format that requires schemas, but does not include the `value_schema` or `value_schema_id` fields
 - Error code 42205 -- Request includes invalid schema.

Example binary request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.binary.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
      "key": "a2V5",
      "value": "Y29uZmx1ZW50"
    },
    {
      "value": "a2Fma2E=",
      "partition": 1
    },
    {
      "value": "bG9ncw=="
    }
  ]
}
```

Copy

Example binary response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 2,
      "offset": 100
    },
    {
      "partition": 1,
      "offset": 101
    },
    {
      "partition": 2,
      "offset": 102
    }
  ]
}
```

```
]
}
```

Copy

Example Avro request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.avro.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "value_schema": "{\"name\":\"int\",\"type\":\"int\"}",
  "records": [
    {
      "value": 12
    },
    {
      "value": 24,
      "partition": 1
    }
  ]
}
```

Copy

Example Avro response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": 32,
  "offsets": [
    {
      "partition": 2,
      "offset": 103
    },
    {
      "partition": 1,
      "offset": 104
    }
  ]
}
```

Copy

Example JSON request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.json.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
```

```

    "key": "somekey",
    "value": {"foo": "bar"}
  },
  {
    "value": [ "foo", "bar" ],
    "partition": 1
  },
  {
    "value": 53.5
  }
]
}

```

Copy

Example JSON response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 2,
      "offset": 100
    },
    {
      "partition": 1,
      "offset": 101
    },
    {
      "partition": 2,
      "offset": 102
    }
  ]
}

```

Copy

6.4 Partitions

The partitions resource provides per-partition metadata, including the current leaders and replicas for each partition. It also allows you to consume and produce messages to single partition using `GET` and `POST` requests.

GET /topics/(string:topic_name)/partitions

Get a list of partitions for the topic.

Parameters:

- **topic_name** (*string*) -- the name of the topic

Response JSON Array of Objects:

- **partition** (*int*) -- ID of the partition
- **leader** (*int*) -- Broker ID of the leader for this partition

- **replicas** (*array*) -- List of brokers acting as replicas for this partition
 - **replicas[i].broker** (*int*) -- Broker ID of the replica
 - **replicas[i].leader** (*boolean*) -- true if this broker is the leader for the partition
 - **replicas[i].in_sync** (*boolean*) -- true if the replica is in sync with the leader
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found

Example request:

```
GET /topics/test/partitions HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

[
  {
    "partition": 1,
    "leader": 1,
    "replicas": [
      {
        "broker": 1,
        "leader": true,
        "in_sync": true,
      },
      {
        "broker": 2,
        "leader": false,
        "in_sync": true,
      },
      {
        "broker": 3,
        "leader": false,
        "in_sync": false,
      }
    ]
  },
  {
    "partition": 2,
    "leader": 2,
    "replicas": [
      {
        "broker": 1,
        "leader": false,
        "in_sync": true,
      },
      {
        "broker": 2,
        "leader": true,

```

```

        "in_sync": true,
      },
      {
        "broker": 3,
        "leader": false,
        "in_sync": false,
      }
    ]
  }
}
]

```

Copy

GET /topics/(string:topic_name)/partitions/(int:partition_id)

Get metadata about a single partition in the topic.

Parameters:

- **topic_name** (*string*) -- Name of the topic
- **partition_id** (*int*) -- ID of the partition to inspect

Response JSON Object:

- **partition** (*int*) -- ID of the partition
- **leader** (*int*) -- Broker ID of the leader for this partition
- **replicas** (*array*) -- List of brokers acting as replicas for this partition
- **replicas[i].broker** (*int*) -- Broker ID of the replica
- **replicas[i].leader** (*boolean*) -- true if this broker is the leader for the partition
- **replicas[i].in_sync** (*boolean*) -- true if the replica is in sync with the leader

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40402 -- Partition not found

Example request:

```

GET /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "partition": 1,
  "leader": 1,
  "replicas": [
    {

```

```

    "broker": 1,
    "leader": true,
    "in_sync": true,
  },
  {
    "broker": 2,
    "leader": false,
    "in_sync": true,
  },
  {
    "broker": 3,
    "leader": false,
    "in_sync": false,
  }
]
}

```

Copy

POST /topics/(string:topic_name)/partitions/(int:partition_id)

Produce messages to one partition of the topic. For the `avro` embedded format, you must provide information about schemas. This may be provided as the full schema encoded as a string, or, after the initial request may be provided as the schema ID returned with the first response.

- Parameters:**
- **topic_name** (*string*) -- Topic to produce the messages to
 - **partition_id** (*int*) -- Partition to produce the messages to

Request JSON Object:

- **key_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **key_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **value_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **value_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **records** -- A list of records to produce to the partition.

Request JSON Array of Objects:

- **records[i].key** (*object*) -- The message key, formatted according to the embedded format, or null to omit a key (optional)
- **records[i].value** (*object*) -- The message value, formatted according to the embedded format

Response JSON Object:

- **key_schema_id** (*int*) -- The ID for the schema used to produce keys, or null if keys were not used

- **value_schema_id** (*int*) -- The ID for the schema used to produce values.

Response JSON Array of Objects:

- **offsets** (*object*) -- List of partitions and offsets the messages were published to
- **offsets[i].partition** (*int*) -- Partition the message was published to. This will be the same as the `partition_id` parameter and is provided only to maintain consistency with responses from producing to a topic
- **offsets[i].offset** (*long*) -- Offset of the message
- **offsets[i].error_code** (*long*) --

An error code classifying the reason this operation failed, or null if it succeeded.

- 1 - Non-retriable Kafka exception
 - 2 - Retriable Kafka exception; the message might be sent successfully if retried
- **offsets[i].error** (*string*) -- An error message describing why the operation failed, or null if it succeeded
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40402 -- Partition not found
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Request includes keys and uses a format that requires schemas, but does not include the `key_schema` or `key_schema_id` fields
 - Error code 42202 -- Request includes values and uses a format that requires schemas, but does not include the `value_schema` or `value_schema_id` fields
 - Error code 42205 -- Request includes invalid schema.

Status Codes:

Example binary request:

```
POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.binary.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
      "key": "a2V5",
      "value": "Y29uZmx1ZW50"
    },
    {
      "value": "a2Fma2E="
    }
  ]
}
```

Copy

Example binary response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,
      "offset": 101,
    }
  ]
}

```

Copy

Example Avro request:

```

POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.avro.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "value_schema": "{\"name\":\"int\",\"type\":\"int\"}"
  "records": [
    {
      "value": 25
    },
    {
      "value": 26
    }
  ]
}

```

Copy

Example Avro response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": 32,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,

```

```

    "offset": 101,
  }
]
}

```

Copy

Example JSON request:

```

POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.json.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
      "key": "somekey",
      "value": {"foo": "bar"}
    },
    {
      "value": 53.5
    }
  ]
}

```

Copy

Example JSON response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,
      "offset": 101,
    }
  ]
}

```

Copy

6.5 Consumers

The consumers resource provides access to the current state of consumer groups, allows you to create a consumer in a consumer group and consume messages from topics and partitions. The proxy can convert data stored in Kafka in serialized form into a JSON-compatible embedded format. Currently three formats are supported: raw binary data is encoded as base64 strings, Avro data is converted into embedded JSON objects, and JSON is embedded directly.

Because consumers are stateful, any consumer instances created with the REST API are tied to a specific REST proxy instance. A full URL is provided when the instance is created and it should be used to construct any subsequent requests. Failing to use the returned URL for future consumer requests will result in *404* errors because the consumer instance will not be found. If a REST proxy instance is shutdown, it will attempt to cleanly destroy any consumers before it is terminated.

POST /consumers/(*string:group_name*)

Create a new consumer instance in the consumer group. The `format` parameter controls the deserialization of data from Kafka and the content type that *must* be used in the `Accept` header of subsequent read API requests performed against this consumer. For example, if the creation request specifies `avro` for the format, subsequent read requests should use `Accept: application/vnd.kafka.avro.v2+json`.

Note that the response includes a URL including the host since the consumer is stateful and tied to a specific REST proxy instance. Subsequent examples in this section use a `Host` header for this specific REST proxy instance.

Parameters:

- **group_name** (*string*) -- The name of the consumer group to join

Request JSON Object:

- **name** (*string*) -- Name for the consumer instance, which will be used in URLs for the consumer. This must be unique, at least within the proxy process handling the request. If omitted, falls back on the automatically generated ID. Using automatically generated names is recommended for most use cases.
- **format** (*string*) -- The format of consumed messages, which is used to convert messages into a JSON-compatible form. Valid values: "binary", "avro", "json". If unspecified, defaults to "binary".
- **auto.offset.reset** (*string*) -- Sets the `auto.offset.reset` setting for the consumer
- **auto.commit.enable** (*string*) -- Sets the `auto.commit.enable` setting for the consumer
- **fetch.min.bytes** (*string*) -- Sets the `fetch.min.bytes` setting for this consumer specifically
- **consumer.request.timeout.ms** (*string*) -- Sets the `consumer.request.timeout.ms` setting for this consumer specifically. This setting controls the maximum total time to wait for messages for a request if the maximum request size has not yet been reached. It does not affect the underlying consumer->broker connection. Default value is taken from the REST proxy config file

Response JSON Object:

- **instance_id** (*string*) -- Unique ID for the consumer instance in this group.
- **base_uri** (*string*) -- Base URI used to construct URIs for subsequent requests against this consumer instance. This will be of the

form `http://hostname:port/consumers/consumer_group/instances/instance_id`.

**Status
Codes:**

- [409 Conflict](#) --
 - Error code 40902 -- Consumer instance with the specified name already exists.
- [422 Unprocessable Entity](#) --
 - Error code 42204 -- Invalid consumer configuration. One of the settings specified in the request contained an invalid value.

Example request:

```
POST /consumers/testgroup/ HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json

{
  "name": "my_consumer",
  "format": "binary",
  "auto.offset.reset": "earliest",
  "auto.commit.enable": "false"
}
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "instance_id": "my_consumer",
  "base_uri": "http://proxy-instance.kafkaproxy.example.com/consumers/testgroup/instances/my_consumer"
}
```

Copy

DELETE /consumers/(string:group_name)/instances/(string:instance)

Destroy the consumer instance.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

Parameters:

- **group_name** (string) -- The name of the consumer group
- **instance** (string) -- The ID of the consumer instance

Status Codes:

- [404 Not Found](#) --

- Error code 40403 -- Consumer instance not found

Example request:

```
DELETE /consumers/testgroup/instances/my_consumer HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

POST /consumers/(string:group_name)/instances/(string:instance)/offsets

Commit a list of offsets for the consumer. When the post body is empty, it commits all the records that have been fetched by the consumer instance.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **offsets** -- A list of offsets to commit for partitions
- **offsets[i].topic** (*string*) -- Name of the topic
- **offsets[i].partition** (*int*) -- Partition ID
- **offset** -- the offset to commit

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
POST /consumers/testgroup/instances/my_consumer/offsets HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json
```

```
{
  "offsets": [
    {
      "topic": "test",
      "partition": 0,
      "offset": 20
    }
  ]
}
```

```
{
  "topic": "test",
  "partition": 1,
  "offset": 30
}
```

Copy

GET /consumers/(*string:group_name*)/instances/(*string:instance*)/offsets

Get the last committed offsets for the given partitions (whether the commit happened by this process or another).

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **partitions** -- A list of partitions to find the last committed offsets for
- **partitions[i].topic** (*string*) -- Name of the topic
- **partitions[i].partition** (*int*) -- Partition ID

Response JSON Array of Objects:

- **offsets** -- A list of committed offsets
- **offsets[i].topic** (*string*) -- Name of the topic for which an offset was committed
- **offsets[i].partition** (*int*) -- Partition ID for which an offset was committed
- **offsets[i].offset** (*int*) -- Committed offset
- **offsets[i].metadata** (*string*) -- Metadata for the committed offset

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40402 -- Partition not found
 - Error code 40403 -- Consumer instance not found

Example request:

```
GET /consumers/testgroup/instances/my_consumer/offsets HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

{
  "partitions": [
    {
      "topic": "test",
      "partition": 0
    },
  ],
}
```

```
{
  "topic": "test",
  "partition": 1
}

]
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{"offsets":
[
  {
    "topic": "test",
    "partition": 0,
    "offset": 21,
    "metadata":""
  },
  {
    "topic": "test",
    "partition": 1,
    "offset": 31,
    "metadata":""
  }
]
}
```

Copy

POST /consumers/(string:group_name)/instances/(string:instance)/subscription

Subscribe to the given list of topics or a topic pattern to get dynamically assigned partitions. If a prior subscription exists, it would be replaced by the latest subscription.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **topics** -- A list of topics to subscribe
- **topics[i].topic** (*string*) -- Name of the topic

Request JSON Object:

- **topic_pattern** (*string*) -- A REGEX pattern. **topics_pattern** and **topics** fields are mutually exclusive.

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

- [409 Conflict](#) --
 - Error code 40903 -- Subscription to topics, partitions and pattern are mutually exclusive.

Example request:

```
POST /consumers/testgroup/instances/my_consumer/subscription HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json

{
  "topics": [
    "test1",
    "test2"
  ]
}
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

Example request:

```
POST /consumers/testgroup/instances/my_consumer/subscription HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json

{
  "topic_pattern": "test.*"
}
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

```
GET /consumers/(<string:group_name>)/instances/(<string:instance>)/subscription
```

Get the current subscribed list of topics.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Response JSON Array of Objects:

- **topics** -- A list of subscribed topics
- **topics[i]** (*string*) -- Name of the topic

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
GET /consumers/testgroup/instances/my_consumer/subscription HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json
```

Copy

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "topics": [
    "test1",
    "test2"
  ]
}
```

Copy

DELETE /consumers/(*string:group_name*)/instances/(*string:instance*)/subscription

Unsubscribe from topics currently subscribed.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
DELETE /consumers/testgroup/instances/my_consumer/subscription HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

HTTP/1.1 204 No Content

Copy

POST /consumers/(*string:group_name*)/instances/(*string:instance*)/assignments

Manually assign a list of partitions to this consumer.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **partitions** -- A list of partitions to assign to this consumer
 - **partitions[i].topic** (*string*) -- Name of the topic
 - **partitions[i].partition** (*int*) -- Partition ID
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found
 - [409 Conflict](#) --
 - Error code 40903 -- Subscription to topics, partitions and pattern are mutually exclusive.

Example request:

```
POST /consumers/testgroup/instances/my_consumer/assignments HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json

{
  "partitions": [
    {
      "topic": "test",
      "partition": 0
    },
    {
      "topic": "test",
      "partition": 1
    }
  ]
}
```

Copy

Example response:

HTTP/1.1 204 No Content

Copy

GET /consumers/(*string:group_name*)/instances/(*string:instance*)/assignments

Get the list of partitions currently manually assigned to this consumer.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Response JSON Array of Objects:

- **partitions** -- A list of partitions manually to assign to this consumer
 - **partitions[i].topic** (*string*) -- Name of the topic
 - **partitions[i].partition** (*int*) -- Partition ID
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
GET /consumers/testgroup/instances/my_consumer/assignments HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json
```

Copy

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "partitions": [
    {
      "topic": "test",
      "partition": 0
    },
    {
      "topic": "test",
      "partition": 1
    }
  ]
}
```

Copy

POST /consumers/(string:group_name)/instances/(string:instance)/positions

Overrides the fetch offsets that the consumer will use for the next set of records to fetch.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **offsets** -- A list of offsets
- **offsets[i].topic** (*string*) -- Name of the topic for
- **offsets[i].partition** (*int*) -- Partition ID
- **offsets[i].offset** (*int*) -- Seek to offset for the next set of records to fetch

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
POST /consumers/testgroup/instances/my_consumer/positions HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json
```

```
{
  "offsets": [
    {
      "topic": "test",
      "partition": 0,
      "offset": 20
    },
    {
      "topic": "test",
      "partition": 1,
      "offset": 30
    }
  ]
}
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

POST /consumers/(*string:group_name*)/instances/(*string:instance*)/positions/beginning

Seek to the first offset for each of the given partitions.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **partitions** -- A list of partitions
- **partitions[i].topic** (*string*) -- Name of the topic
- **partitions[i].partition** (*int*) -- Partition ID

- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
POST /consumers/testgroup/instances/my_consumer/positions/beginning HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Content-Type: application/vnd.kafka.v2+json

{
  "partitions": [
    {
      "topic": "test",
      "partition": 0
    },
    {
      "topic": "test",
      "partition": 1
    }
  ]
}
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

```
POST /consumers/(string:group_name)/instances/(string:instance)/positions/end
```

Seek to the last offset for each of the given partitions.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Request JSON Array of Objects:

- **partitions** -- A list of partitions
 - **partitions[i].topic** (*string*) -- Name of the topic
 - **partitions[i].partition** (*int*) -- Partition ID
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
POST /consumers/testgroup/instances/my_consumer/positions/end HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
```

Content-Type: application/vnd.kafka.v2+json

```
{
  "partitions": [
    {
      "topic": "test",
      "partition": 0
    },
    {
      "topic": "test",
      "partition": 1
    }
  ]
}
```

Copy

Example response:

HTTP/1.1 204 No Content

Copy

GET /consumers/(string:group_name)/instances/(string:instance)/records

Fetch data for the topics or partitions specified using one of the subscribe/assign APIs.

The format of the embedded data returned by this request is determined by the format specified in the initial consumer instance creation request and must match the format of the `Accept` header.

Mismatches will result in error code `40601`.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- Parameters:
- **group_name** (string) -- The name of the consumer group
 - **instance** (string) -- The ID of the consumer instance

Query Parameters:

- **timeout** -- Maximum amount of milliseconds the REST proxy will spend fetching records. Other parameters controlling actual time spent fetching records: *max_bytes* and *fetch.min.bytes*. Default value is undefined. This parameter is used only if it's smaller than the *consumer.timeout.ms* that is defined either during consumer instance creation or in the proxy's config file.
- **max_bytes** -- The maximum number of bytes of unencoded keys and values that should be included in the response. This provides approximate control over the size of responses and the amount of memory required to store the decoded response. The actual limit will be the minimum of this setting and the server-side configuration `consumer.request.max.bytes`. Default is unlimited.

Response JSON Array of Objects:

- **topic** (*string*) -- The topic
- **key** (*string*) -- The message key, formatted according to the embedded format
- **value** (*string*) -- The message value, formatted according to the embedded format
- **partition** (*int*) -- Partition of the message
- **offset** (*long*) -- Offset of the message

Status Codes:

- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found
- [406 Not Acceptable](#) --
 - Error code 40601 -- Consumer format does not match the embedded format requested by the **Accept** header.

Example binary request:

```
GET /consumers/testgroup/instances/my_consumer/records?timeout=3000&max_bytes=300000 HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.binary.v2+json
```

Copy

Example binary response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.binary.v2+json
```

```
[
  {
    "topic": "test",
    "key": "a2V5",
    "value": "Y29uZmx1ZW50",
    "partition": 1,
    "offset": 100,
  },
  {
    "topic": "test",
    "key": "a2V5",
    "value": "a2Fma2E=",
    "partition": 2,
    "offset": 101,
  }
]
```

Copy

Example Avro request:

```
GET /consumers/avrogroup/instances/my_avro_consumer/records?timeout=3000&max_bytes=300000 HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
```

Accept: application/vnd.kafka.avro.v2+json

Copy

Example Avro response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.avro.v2+json

```
[
  {
    "topic": "test",
    "key": 1,
    "value": {
      "id": 1,
      "name": "Bill"
    },
    "partition": 1,
    "offset": 100,
  },
  {
    "topic": "test",
    "key": 2,
    "value": {
      "id": 2,
      "name": "Melinda"
    },
    "partition": 2,
    "offset": 101,
  }
]
```

Copy

Example JSON request:

GET /consumers/jsongroup/instances/my_json_consumer/records?timeout=3000&max_bytes=300000 **HTTP/1.1**
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.json.v2+json

Copy

Example JSON response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.json.v2+json

```
[
  {
    "topic": "test",
    "key": "somekey",
    "value": {"foo": "bar"},
    "partition": 1,
    "offset": 10,
  },
  {
    "topic": "test",
    "key": "somekey",
```

```

    "value": ["foo", "bar"],
    "partition": 2,
    "offset": 11,
  }
]

```

Copy

6.6 Brokers

The brokers resource provides access to the current state of Kafka brokers in the cluster.

GET /brokers

Get a list of brokers.

Response JSON Object:

- **brokers** (*array*) -- List of broker IDs

Example request:

```

GET /brokers HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/json

```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v2+json

{
  "brokers": [1, 2, 3]
}

```

Copy

6.7 Topics

The topics resource provides information about the topics in your Kafka cluster and their current state. It also lets you produce messages by making **POST** requests to specific topics.

GET /topics

Get a list of Kafka topics.

Response JSON Object:

- **topics** (*array*) -- List of topic names

Example request:

```
GET /topics HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

["topic1", "topic2"]
```

Copy

GET /topics/(*string:topic_name*)

Get metadata about a specific topic.

Parameters:

- **topic_name** (*string*) -- Name of the topic to get metadata about

Response JSON Object:

- **name** (*string*) -- Name of the topic
- **configs** (*map*) -- Per-topic configuration overrides
- **partitions** (*array*) -- List of partitions for this topic
- **partitions[i].partition** (*int*) -- the ID of this partition
- **partitions[i].leader** (*int*) -- the broker ID of the leader for this partition
- **partitions[i].replicas** (*array*) -- list of replicas for this partition, including the leader
- **partitions[i].replicas[j].broker** (*array*) -- broker ID of the replica
- **partitions[i].replicas[j].leader** (*boolean*) -- true if this replica is the leader for the partition
- **partitions[i].replicas[j].in_sync** (*boolean*) -- true if this replica is currently in sync with the leader

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found

Example request:

```
GET /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

```
{
  "name": "test",
  "configs": {
    "cleanup.policy": "compact"
  },
  "partitions": [
    {
      "partition": 1,
      "leader": 1,
      "replicas": [
        {
          "broker": 1,
          "leader": true,
          "in_sync": true,
        },
        {
          "broker": 2,
          "leader": false,
          "in_sync": true,
        }
      ]
    },
    {
      "partition": 2,
      "leader": 2,
      "replicas": [
        {
          "broker": 1,
          "leader": false,
          "in_sync": true,
        },
        {
          "broker": 2,
          "leader": true,
          "in_sync": true,
        }
      ]
    }
  ]
}
```

Copy

POST /topics/(string:topic_name)

Produce messages to a topic, optionally specifying keys or partitions for the messages. If no partition is provided, one will be chosen based on the hash of the key. If no key is provided, the partition will be chosen for each message in a round-robin fashion.

We currently support Avro, JSON and binary message formats.

For the `avro` embedded format, you must provide information about schemas and the REST proxy must be configured with the URL to access Schema Registry (`schema.registry.url`). Schemas may be provided as the full schema encoded as a string, or, after the initial request may be provided as the schema ID returned with the first response. Note that if you use Avro for value you must also use Avro for the key, but the key and value may have different schemas.

Parameters:

- **topic_name** (*string*) -- Name of the topic to produce the messages to

Request JSON Object:

- **key_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data). This is only needed for Avro format.
- **key_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **value_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data). This is only needed for Avro format.
- **value_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.

Request JSON Array of Objects:

- **records** -- A list of records to produce to the topic.
- **records[i].key** (*object*) -- The message key, formatted according to the embedded format, or null to omit a key (optional)
- **records[i].value** (*object*) -- The message value, formatted according to the embedded format
- **records[i].partition** (*int*) -- Partition to store the message in (optional)

Response JSON Object:

- **key_schema_id** (*int*) -- The ID for the schema used to produce keys, or null if keys were not used
- **value_schema_id** (*int*) -- The ID for the schema used to produce values.

Response JSON Array of Objects:

- **offsets** (*object*) -- List of partitions and offsets the messages were published to
- **offsets[i].partition** (*int*) -- Partition the message was published to, or null if publishing the message failed
- **offsets[i].offset** (*long*) -- Offset of the message, or null if publishing the message failed
- **offsets[i].error_code** (*long*) --

An error code classifying the reason this operation failed, or null if it succeeded.

- 1 - Non-retriable Kafka exception
- 2 - Retriable Kafka exception; the message might be sent successfully if retried
- **offsets[i].error** (*string*) -- An error message describing why the operation failed, or null if it succeeded

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Request includes keys and uses a format that requires schemas, but does not include the `key_schema` or `key_schema_id` fields
 - Error code 42202 -- Request includes values and uses a format that requires schemas, but does not include the `value_schema` or `value_schema_id` fields
 - Error code 42205 -- Request includes invalid schema.

Example binary request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.binary.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
      "key": "a2V5",
      "value": "Y29uZmx1ZW50"
    },
    {
      "value": "a2Fma2E=",
      "partition": 1
    },
    {
      "value": "bG9ncw=="
    }
  ]
}
```

Copy

Example binary response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 2,
      "offset": 100
    },
    {
      "partition": 1,
      "offset": 101
    },
    {
      "partition": 2,
      "offset": 102
    }
  ]
}
```

```
]
}
```

Copy

Example Avro request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.avro.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

{
  "value_schema": "{\"name\":\"int\",\"type\":\"int\"}"
  "records": [
    {
      "value": 12
    },
    {
      "value": 24,
      "partition": 1
    }
  ]
}
```

Copy

Example Avro response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "key_schema_id": null,
  "value_schema_id": 32,
  "offsets": [
    {
      "partition": 2,
      "offset": 103
    },
    {
      "partition": 1,
      "offset": 104
    }
  ]
}
```

Copy

Example JSON request:

```
POST /topics/test HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.json.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

{
  "records": [
    {
```

```

    "key": "somekey",
    "value": {"foo": "bar"}
  },
  {
    "value": [ "foo", "bar" ],
    "partition": 1
  },
  {
    "value": 53.5
  }
]
}

```

Copy

Example JSON response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 2,
      "offset": 100
    },
    {
      "partition": 1,
      "offset": 101
    },
    {
      "partition": 2,
      "offset": 102
    }
  ]
}

```

Copy

6.8 Partitions

The partitions resource provides per-partition metadata, including the current leaders and replicas for each partition. It also allows you to consume and produce messages to single partition using `GET` and `POST` requests.

GET /topics/(string:topic_name)/partitions

Get a list of partitions for the topic.

Parameters:

- **topic_name** (*string*) -- the name of the topic

Response JSON Array of Objects:

- **partition** (*int*) -- ID of the partition
- **leader** (*int*) -- Broker ID of the leader for this partition

- **replicas** (*array*) -- List of brokers acting as replicas for this partition
 - **replicas[i].broker** (*int*) -- Broker ID of the replica
 - **replicas[i].leader** (*boolean*) -- true if this broker is the leader for the partition
 - **replicas[i].in_sync** (*boolean*) -- true if the replica is in sync with the leader
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found

Example request:

```
GET /topics/test/partitions HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

[
  {
    "partition": 1,
    "leader": 1,
    "replicas": [
      {
        "broker": 1,
        "leader": true,
        "in_sync": true,
      },
      {
        "broker": 2,
        "leader": false,
        "in_sync": true,
      },
      {
        "broker": 3,
        "leader": false,
        "in_sync": false,
      }
    ]
  },
  {
    "partition": 2,
    "leader": 2,
    "replicas": [
      {
        "broker": 1,
        "leader": false,
        "in_sync": true,
      },
      {
        "broker": 2,
        "leader": true,
        "in_sync": true,
      }
    ]
  }
]
```



```

    },
    {
      "broker": 3,
      "leader": false,
      "in_sync": false,
    }
  ]
}
]

```

Copy

GET /topics/(string:topic_name)/partitions/(int:partition_id)

Get metadata about a single partition in the topic.

Parameters:

- **topic_name** (*string*) -- Name of the topic
- **partition_id** (*int*) -- ID of the partition to inspect

Response JSON Object:

- **partition** (*int*) -- ID of the partition
- **leader** (*int*) -- Broker ID of the leader for this partition
- **replicas** (*array*) -- List of brokers acting as replicas for this partition
- **replicas[i].broker** (*int*) -- Broker ID of the replica
- **replicas[i].leader** (*boolean*) -- true if this broker is the leader for the partition
- **replicas[i].in_sync** (*boolean*) -- true if the replica is in sync with the leader

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40402 -- Partition not found

Example request:

```

GET /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "partition": 1,
  "leader": 1,
  "replicas": [
    {
      "broker": 1,

```

```

    "leader": true,
    "in_sync": true,
  },
  {
    "broker": 2,
    "leader": false,
    "in_sync": true,
  },
  {
    "broker": 3,
    "leader": false,
    "in_sync": false,
  }
]
}

```

Copy

```
GET /topics/(<string:topic_name>)/partitions/(<int:partition_id>)/messages?offset=(<int>)[&count=(<int>)]
```

Consume messages from one partition of the topic.

Parameters:

- **topic_name** (*string*) -- Topic to consume the messages from
- **partition_id** (*int*) -- Partition to consume the messages from

Query Parameters:

- **offset** (*int*) -- Offset to start from
- **count** (*int*) -- Number of messages to consume (optional). Default is 1.

Response JSON Array of Objects:

- **key** (*string*) -- The message key, formatted according to the embedded format
 - **value** (*string*) -- The message value, formatted according to the embedded format
 - **partition** (*int*) -- Partition of the message
 - **offset** (*long*) -- Offset of the message
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40402 -- Partition not found
 - Error code 40404 -- Leader not available
 - [500 Internal Server Error](#) --
 - Error code 500 -- General consumer error response, caused by an exception during the operation. An error message is included in the standard format which explains the cause.
 - [503 Service Unavailable](#) --
 - Error code 50301 -- No SimpleConsumer is available at the time in the pool. The request can be retried. You can increase the pool size or the pool timeout to avoid this error in the future.

Example binary request:

```
GET /topic/test/partitions/1/messages?offset=10&count=2 HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.binary.v1+json
```

Copy

Example binary response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.binary.v1+json
```

```
[
  {
    "key": "a2V5",
    "value": "Y29uZmx1ZW50",
    "partition": 1,
    "offset": 10,
  },
  {
    "key": "a2V5",
    "value": "a2Fma2E=",
    "partition": 1,
    "offset": 11,
  }
]
```

Copy

Example Avro request:

```
GET /topic/test/partitions/1/messages?offset=1 HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.avro.v1+json
```

Copy

Example Avro response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.avro.v1+json
```

```
[
  {
    "key": 1,
    "value": {
      "id": 1,
      "name": "Bill"
    },
    "partition": 1,
    "offset": 1,
  }
]
```

Copy

Example JSON request:

```
GET /topic/test/partitions/1/messages?offset=10&count=2 HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.json.v1+json
```

Copy

Example JSON response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.json.v1+json

[
  {
    "key": "somekey",
    "value": {"foo": "bar"},
    "partition": 1,
    "offset": 10,
  },
  {
    "key": "somekey",
    "value": ["foo", "bar"],
    "partition": 1,
    "offset": 11,
  }
]
```

Copy

POST /topics/(string:topic_name)/partitions/(int:partition_id)

Produce messages to one partition of the topic. For the `avro` embedded format, you must provide information about schemas. This may be provided as the full schema encoded as a string, or, after the initial request may be provided as the schema ID returned with the first response.

- Parameters:**
- **topic_name** (*string*) -- Topic to produce the messages to
 - **partition_id** (*int*) -- Partition to produce the messages to

Request JSON Object:

- **key_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **key_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **value_schema** (*string*) -- Full schema encoded as a string (e.g. JSON serialized for Avro data)
- **value_schema_id** (*int*) -- ID returned by a previous request using the same schema. This ID corresponds to the ID of the schema in the registry.
- **records** -- A list of records to produce to the partition.

Request JSON Array of Objects:

- **records[i].key** (*object*) -- The message key, formatted according to the embedded format, or null to omit a key (optional)
- **records[i].value** (*object*) -- The message value, formatted according to the embedded format

Response JSON Object:

- **key_schema_id** (*int*) -- The ID for the schema used to produce keys, or null if keys were not used
- **value_schema_id** (*int*) -- The ID for the schema used to produce values.

Response JSON Array of Objects:

- **offsets** (*object*) -- List of partitions and offsets the messages were published to
- **offsets[i].partition** (*int*) -- Partition the message was published to. This will be the same as the `partition_id` parameter and is provided only to maintain consistency with responses from producing to a topic
- **offsets[i].offset** (*long*) -- Offset of the message
- **offsets[i].error_code** (*long*) --

An error code classifying the reason this operation failed, or null if it succeeded.

- 1 - Non-retriable Kafka exception
- 2 - Retriable Kafka exception; the message might be sent successfully if retried
- **offsets[i].error** (*string*) -- An error message describing why the operation failed, or null if it succeeded

- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40402 -- Partition not found
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Request includes keys and uses a format that requires schemas, but does not include the `key_schema` or `key_schema_id` fields
 - Error code 42202 -- Request includes values and uses a format that requires schemas, but does not include the `value_schema` or `value_schema_id` fields
 - Error code 42205 -- Request includes invalid schema.

Status Codes:

Example binary request:

```
POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.binary.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

```
{
  "records": [
    {
      "key": "a2V5",
      "value": "Y29uZmx1ZW50"
    },
    {
      "value": "a2Fma2E="
    }
  ]
}
```

Copy

Example binary response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

```
{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,
      "offset": 101,
    }
  ]
}
```

Copy

Example Avro request:

POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.avro.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

```
{
  "value_schema": "{\"name\":\"int\",\"type\":\"int\"}"
  "records": [
    {
      "value": 25
    },
    {
      "value": 26
    }
  ]
}
```

Copy

Example Avro response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

```
{
  "key_schema_id": null,
  "value_schema_id": 32,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,
      "offset": 101,
    }
  ]
}
```

Copy

Example JSON request:

POST /topics/test/partitions/1 HTTP/1.1
Host: kafkaproxy.example.com
Content-Type: application/vnd.kafka.json.v1+json
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

```
{
  "records": [
    {
      "key": "somekey",
      "value": {"foo": "bar"}
    },
    {
      "value": 53.5
    }
  ]
}
```

Copy

Example JSON response:

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

```
{
  "key_schema_id": null,
  "value_schema_id": null,
  "offsets": [
    {
      "partition": 1,
      "offset": 100,
    },
    {
      "partition": 1,
      "offset": 101,
    }
  ]
}
```


}

Copy

6.9 Consumers

The consumers resource provides access to the current state of consumer groups, allows you to create a consumer in a consumer group and consume messages from topics and partitions. The proxy can convert data stored in Kafka in serialized form into a JSON-compatible embedded format. Currently three formats are supported: raw binary data is encoded as base64 strings, Avro data is converted into embedded JSON objects, and JSON is embedded directly.

Because consumers are stateful, any consumer instances created with the REST API are tied to a specific REST proxy instance. A full URL is provided when the instance is created and it should be used to construct any subsequent requests. Failing to use the returned URL for future consumer requests will result in *404* errors because the consumer instance will not be found. If a REST proxy instance is shutdown, it will attempt to cleanly destroy any consumers before it is terminated.

Consumers may not change the set of topics they are subscribed to once they have started consuming messages. For example, if a consumer is created without specifying topic subscriptions, the first read from a topic will subscribe the consumer to that topic and attempting to read from another topic will cause an error.

POST /consumers/(*string:group_name*)

Create a new consumer instance in the consumer group. The `format` parameter controls the deserialization of data from Kafka and the content type that *must* be used in the `Accept` header of subsequent read API requests performed against this consumer. For example, if the creation request specifies `avro` for the format, subsequent read requests should use `Accept: application/vnd.kafka.avro.v1+json`.

Note that the response includes a URL including the host since the consumer is stateful and tied to a specific REST proxy instance. Subsequent examples in this section use a `Host` header for this specific REST proxy instance.

Parameters:

- **group_name** (*string*) -- The name of the consumer group to join

Request JSON Object:

- **id** (*string*) -- **DEPRECATED** Unique ID for the consumer instance in this group. If omitted, one will be automatically generated
- **name** (*string*) -- Name for the consumer instance, which will be used in URLs for the consumer. This must be unique, at least within the proxy process handling the request. If omitted, falls back on the automatically generated ID. Using automatically generated names is recommended for most use cases.
- **format** (*string*) -- The format of consumed messages, which is used to convert messages into a JSON-compatible form. Valid values: "binary", "avro", "json". If unspecified, defaults to "binary".
- **auto.offset.reset** (*string*) -- Sets the `auto.offset.reset` setting for the consumer

- **auto.commit.enable** (*string*) -- Sets the `auto.commit.enable` setting for the consumer

Response JSON Object:

- **instance_id** (*string*) -- Unique ID for the consumer instance in this group. If provided in the initial request, this will be identical to `id`.
- **base_uri** (*string*) -- Base URI used to construct URIs for subsequent requests against this consumer instance. This will be of the form `http://hostname:port/consumers/consumer_group/instances/instance_id`.

Status Codes:

- [409 Conflict](#) --
 - Error code 40902 -- Consumer instance with the specified name already exists.
- [422 Unprocessable Entity](#) --
 - Error code 42204 -- Invalid consumer configuration. One of the settings specified in the request contained an invalid value.

Example request:

```
POST /consumers/testgroup/ HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

{
  "name": "my_consumer",
  "format": "binary",
  "auto.offset.reset": "smallest",
  "auto.commit.enable": "false"
}
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "instance_id": "my_consumer",
  "base_uri": "http://proxy-instance.kafkaproxy.example.com/consumers/testgroup/instances/my_consumer"
}
```

Copy

POST /consumers/(string:group_name)/instances/(string:instance)/offsets

Commit offsets for the consumer. Returns a list of the partitions with the committed offsets.

The body of this request is empty. The offsets are determined by the current state of the consumer instance on the proxy. The returned state includes both `consumed` and `committed` offsets. After a successful commit, these should be identical; however, both are included so the output format is consistent with other API calls that return the offsets.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- Parameters:**
- **group_name** (*string*) -- The name of the consumer group
 - **instance** (*string*) -- The ID of the consumer instance

Response JSON Array of Objects:

- **topic** (*string*) -- Name of the topic for which an offset was committed
 - **partition** (*int*) -- Partition ID for which an offset was committed
 - **consumed** (*long*) -- The offset of the most recently consumed message
 - **committed** (*long*) -- The committed offset value. If the commit was successful, this should be identical to `consumed`.
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40403 -- Consumer instance not found

Example request:

```
POST /consumers/testgroup/instances/my_consumer/offsets HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

[
  {
    "topic": "test",
    "partition": 1,
    "consumed": 100,
    "committed": 100
  },
  {
    "topic": "test",
    "partition": 2,
    "consumed": 200,
    "committed": 200
  },
  {
    "topic": "test2",
    "partition": 1,
    "consumed": 50,
```

```

    "committed": 50
  }
]

```

Copy

DELETE /consumers/(string:group_name)/instances/(string:instance)

Destroy the consumer instance.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- | | |
|----------------------|---|
| Parameters: | <ul style="list-style-type: none"> • group_name (<i>string</i>) -- The name of the consumer group • instance (<i>string</i>) -- The ID of the consumer instance |
| Status Codes: | <ul style="list-style-type: none"> • 404 Not Found -- <ul style="list-style-type: none"> ◦ Error code 40403 -- Consumer instance not found |

Example request:

```

DELETE /consumers/testgroup/instances/my_consumer HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json

```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

GET /consumers/(string:group_name)/instances/(string:instance)/topics/(string:topic_name)

Consume messages from a topic. If the consumer is not yet subscribed to the topic, this adds it as a subscriber, possibly causing a consumer rebalance.

The format of the embedded data returned by this request is determined by the format specified in the initial consumer instance creation request and must match the format of the **Accept** header.

Mismatches will result in error code **40601**.

Note that this request *must* be made to the specific REST proxy instance holding the consumer instance.

- | | |
|--------------------|---|
| Parameters: | <ul style="list-style-type: none"> • group_name (<i>string</i>) -- The name of the consumer group • instance (<i>string</i>) -- The ID of the consumer instance • topic_name (<i>string</i>) -- The topic to consume messages from. |
|--------------------|---|

Query Parameters:

- **max_bytes** -- The maximum number of bytes of unencoded keys and values that should be included in the response. This provides approximate control over the size of responses and the amount of memory required to store the decoded response. The actual limit will be the minimum of this setting and the server-side configuration `consumer.request.max.bytes`. Default is unlimited.

Response JSON Array of Objects:

- **key** (*string*) -- The message key, formatted according to the embedded format
- **value** (*string*) -- The message value, formatted according to the embedded format
- **partition** (*int*) -- Partition of the message
- **offset** (*long*) -- Offset of the message
- [404 Not Found](#) --
 - Error code 40401 -- Topic not found
 - Error code 40403 -- Consumer instance not found
- [406 Not Acceptable](#) --
 - Error code 40601 -- Consumer format does not match the embedded format requested by the `Accept` header.
- [409 Conflict](#) --
 - Error code 40901 -- Consumer has already initiated a subscription. Consumers may subscribe to multiple topics, but all subscriptions must be initiated in a single request.
- [500 Internal Server Error](#) --
 - Error code 500 -- General consumer error response, caused by an exception during the operation. An error message is included in the standard format which explains the cause.

Status Codes:**Example binary request:**

```
GET /consumers/testgroup/instances/my_consumer/topics/test_topic HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.binary.v1+json
```

Copy

Example binary response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.binary.v1+json

[
  {
    "key": "a2V5",
    "value": "Y29uZmx1ZW50",
    "partition": 1,
```

```

    "offset": 100,
    "topic": "test_topic"
  },
  {
    "key": "a2V5",
    "value": "a2Fma2E=",
    "partition": 2,
    "offset": 101,
    "topic": "test_topic"
  }
]

```

Copy

Example Avro request:

```

GET /consumers/avrogroup/instances/my_avro_consumer/topics/test_avro_topic HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.avro.v1+json

```

Copy

Example Avro response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.avro.v1+json

```

```

[
  {
    "key": 1,
    "value": {
      "id": 1,
      "name": "Bill"
    },
    "partition": 1,
    "offset": 100,
    "topic": "test_avro_topic"
  },
  {
    "key": 2,
    "value": {
      "id": 2,
      "name": "Melinda"
    },
    "partition": 2,
    "offset": 101,
    "topic": "test_avro_topic"
  }
]

```

Copy

Example JSON request:

```

GET /consumers/jsongroup/instances/my_json_consumer/topics/test_json_topic HTTP/1.1
Host: proxy-instance.kafkaproxy.example.com
Accept: application/vnd.kafka.json.v1+json

```

Copy

Example JSON response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.json.v1+json
```

```
[
  {
    "key": "somekey",
    "value": {"foo": "bar"},
    "partition": 1,
    "offset": 10,
    "topic": "test_json_topic"
  },
  {
    "key": "somekey",
    "value": ["foo", "bar"],
    "partition": 2,
    "offset": 11,
    "topic": "test_json_topic"
  }
]
```

Copy

6.10 Brokers

The brokers resource provides access to the current state of Kafka brokers in the cluster.

GET /brokers

Get a list of brokers.

Response JSON Object:

- **brokers** (*array*) -- List of broker IDs

Example request:

```
GET /brokers HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.kafka.v1+json, application/vnd.kafka+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.kafka.v1+json

{
  "brokers": [1, 2, 3]
}
```

Please note that all the material included in APPENDIX A is taken from its online source found in [22].

7 APPENDIX B to ANNEX

In this appendix the Schema Registry API Reference is quoted as it appears in the relative online resource [23] for convenience purposes and as requested:

The material in this appendix is Copyrighted by Confluent, Inc (© Copyright 2019, Confluent, Inc).

7.1 Compatibility

The Schema Registry server can enforce certain compatibility rules when new schemas are registered in a subject. These are the compatibility types:

- `BACKWARD`: (*default*) consumers using the new schema can read data written by producers using the latest registered schema
- `BACKWARD_TRANSITIVE`: consumers using the new schema can read data written by producers using all previously registered schemas
- `FORWARD`: consumers using the latest registered schema can read data written by producers using the new schema
- `FORWARD_TRANSITIVE`: consumers using all previously registered schemas can read data written by producers using the new schema
- `FULL`: the new schema is forward and backward compatible with the latest registered schema
- `FULL_TRANSITIVE`: the new schema is forward and backward compatible with all previously registered schemas
- `NONE`: schema compatibility checks are disabled

We recommend keeping the default backward compatibility since it's common to have all data loaded into Hadoop.

For more details on Avro schema resolution, see [Schema Evolution and Compatibility](#).

7.2 Content Types

The Schema Registry REST server uses content types for both requests and responses to indicate the serialization format of the data as well as the version of the API being used. Currently, the only serialization format supported is JSON and the only version of the API is `v1`. However, to remain compatible with future versions, you *should* specify preferred content types in requests and check the content types of responses.

The preferred format for content types is `application/vnd.schemaregistry.v1+json`, where `v1` is the API version and `json` is the serialization format. However, other less specific content types are permitted, including `application/vnd.schemaregistry+json` to indicate no specific API version should be used (the most recent stable version will be used), `application/json`, and `application/octet-stream`. The latter two are only supported for compatibility and ease of use.

Your requests *should* specify the most specific format and version information possible via the HTTP `Accept` header:

```
Accept: application/vnd.schemaregistry.v1+json
```

Copy

The server also supports content negotiation, so you may include multiple, weighted preferences:


```
Accept: application/vnd.schemaregistry.v1+json; q=0.9, application/json; q=0.5
```

Copy

which can be useful when, for example, a new version of the API is preferred but you cannot be certain it is available yet.

7.3 Errors

All API endpoints use a standard error message format for any requests that return an HTTP status indicating an error (any 400 or 500 statuses). For example, a request entity that omits a required field may generate the following response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.schemaregistry.v1+json

{
  "error_code": 422,
  "message": "schema may not be empty"
}
```

Copy

Although it is good practice to check the status code, you may safely parse the response of any non-DELETE API calls and check for the presence of an `error_code` field to detect errors.

7.4 Schemas

GET /schemas/ids/{int: id}

Get the schema string identified by the input ID.

Parameters:

- **id** (*int*) -- the globally unique identifier of the schema

Response JSON Object:

- **schema** (*string*) -- Schema string identified by the ID

Status Codes:

- [404 Not Found](#) --
 - Error code 40403 -- Schema not found
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend datastore

Example request:

```
GET /schemas/ids/1 HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "schema": "{\"type\": \"string\"}"
}
```

Copy

7.5 Subjects

The subjects resource provides a list of all registered subjects in your Schema Registry. A subject refers to the name under which the schema is registered. If you are using Schema Registry for Kafka, then a subject refers to either a "<topic>-key" or "<topic>-value" depending on whether you are registering the key schema for that topic or the value schema.

GET /subjects

Get a list of registered subjects.

Response JSON Array of Objects:

- **name** (*string*) -- Subject
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend datastore

Status Codes:
Example request:

```
GET /subjects HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

["subject1", "subject2"]
```

Copy

GET /subjects/(string: subject)/versions

Get a list of versions registered under the specified subject.

Parameters:

- **subject** (*string*) -- the name of the subject

Response JSON Array of Objects:

- **version** (*int*) -- version of the schema registered under this subject
- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
- Status Codes:**
 - [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend datastore

Example request:

```
GET /subjects/test/versions HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

[
  1, 2, 3, 4
]
```

Copy

DELETE /subjects/(string: subject)

Deletes the specified subject and its associated compatibility level if registered. It is recommended to use this API only when a topic needs to be recycled or in development environment.

Parameters:

- **subject** (*string*) -- the name of the subject

Response JSON Array of Objects:

- **version** (*int*) -- version of the schema deleted under this subject
- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
- Status Codes:**
 - [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend datastore

Example request:

```
DELETE /subjects/test HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

[
  1, 2, 3, 4
]
```

Copy

GET /subjects/(string: subject)/versions/(versionId: version)

Get a specific version of the schema registered under this subject

Parameters:

- **subject** (*string*) -- Name of the subject
- **version** (*versionId*) -- Version of the schema to be returned. Valid values for versionId are between [1,2^31-1] or the string "latest". "latest" returns the last registered schema under the specified subject. Note that there may be a new latest schema that gets registered right after this request is served.

Response JSON Object:

- **subject** (*string*) -- Name of the subject that this schema is registered under
 - **id** (*int*) -- Globally unique identifier of the schema
 - **version** (*int*) -- Version of the returned schema
 - **schema** (*string*) -- The Avro schema string
-
- **Status Codes:**
 - [404 Not Found](#) --
 - Error code 40401 -- Subject not found
 - Error code 40402 -- Version not found
 - [422 Unprocessable Entity](#) --
 - Error code 42202 -- Invalid version
 - [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Example request:

```
GET /subjects/test/versions/1 HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "name": "test",
  "version": 1,
  "schema": "{\"type\": \"string\"}"
}

```

Copy

GET /subjects/(string: subject)/versions/(versionId: version)/schema

Get the avro schema for the specified version of this subject. The unescaped schema only is returned.

Parameters:

- **subject** (*string*) -- Name of the subject
- **version** (*versionId*) -- Version of the schema to be returned. Valid values for versionId are between [1,2^31-1] or the string "latest". "latest" returns the last registered schema under the specified subject. Note that there may be a new latest schema that gets registered right after this request is served.

Response JSON Object:

- **schema** (*string*) -- The Avro schema string (unescaped)
- Status Codes:**
- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
 - Error code 40402 -- Version not found
 - [422 Unprocessable Entity](#) --
 - Error code 42202 -- Invalid version
 - [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Example request:

```

GET /subjects/test/versions/1/schema HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

```

```
{"type": "string"}
```

Copy

POST /subjects/(string: subject)/versions

Register a new schema under the specified subject. If successfully registered, this returns the unique identifier of this schema in the registry. The returned identifier should be used to retrieve this schema from the schemas resource and is different from the schema's version which is associated with the subject. If the same schema is registered under a different subject, the same identifier will be returned. However, the version of the schema may be different under different subjects.

A schema should be compatible with the previously registered schema or schemas (if there are any) as per the configured compatibility level. The configured compatibility level can be obtained by issuing a `GET http:get:: /config/(string: subject)`. If that returns null, then `GET http:get:: /config`

When there are multiple instances of Schema Registry running in the same cluster, the schema registration request will be forwarded to one of the instances designated as the primary. If the primary is not available, the client will get an error code indicating that the forwarding has failed.

Parameters:

- **subject** (*string*) -- Subject under which the schema will be registered

Request JSON Object:

- **schema** -- The Avro schema string

Status Codes:

- [409 Conflict](#) -- Incompatible Avro schema
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Invalid Avro schema
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store
 - Error code 50002 -- Operation timed out
 - Error code 50003 -- Error while forwarding the request to the primary

Example request:

```
POST /subjects/test/versions HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

{
  "schema":
  "{
    \"type\": \"record\",
    \"name\": \"test\",
    \"fields\":
    [
      {
        \"type\": \"string\",
        \"name\": \"field1\"
      },
    ]
  }"
```

```

    {
      \"type\": \"int\",
      \"name\": \"field2\"
    }
  ]
}

```

Copy

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{"id":1}

```

Copy

POST /subjects/(string: subject)

Check if a schema has already been registered under the specified subject. If so, this returns the schema string along with its globally unique identifier, its version under this subject and the subject name.

Parameters:

- **subject** (*string*) -- Subject under which the schema will be registered

Response JSON Object:

- **subject** (*string*) -- Name of the subject that this schema is registered under
- **id** (*int*) -- Globally unique identifier of the schema
- **version** (*int*) -- Version of the returned schema
- **schema** (*string*) -- The Avro schema string

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
 - Error code 40403 -- Schema not found
- [500 Internal Server Error](#) -- Internal server error

Example request:

```

POST /subjects/test HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

{
  "schema":
    "{
      \"type\": \"record\",
      \"name\": \"test\",
      \"fields\":

```

```

    [
      {
        \"type\": \"string\",
        \"name\": \"field1\"
      },
      {
        \"type\": \"int\",
        \"name\": \"field2\"
      }
    ]
  }
}

```

Copy

Example response:

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

```

{
  "subject": "test",
  "id": 1
  "version": 3
  "schema":
    "{
      \"type\": \"record\",
      \"name\": \"test\",
      \"fields\":
        [
          {
            \"type\": \"string\",
            \"name\": \"field1\"
          },
          {
            \"type\": \"int\",
            \"name\": \"field2\"
          }
        ]
    }"
}

```

Copy

DELETE /subjects/(string: subject)/versions/(versionId: version)

Deletes a specific version of the schema registered under this subject. This only deletes the version and the schema ID remains intact making it still possible to decode data using the schema ID. This API is recommended to be used only in development environments or under extreme circumstances where-in, its required to delete a previously registered schema for compatibility purposes or re-register previously registered schema.

Parameters:

- **subject** (*string*) -- Name of the subject
- **version** (*versionId*) -- Version of the schema to be deleted. Valid values for versionId are between [1,2^31-1] or the string "latest". "latest" deletes the last registered schema under the specified subject. Note that there

may be a new latest schema that gets registered right after this request is served.

Response JSON Object:

- **int** -- Version of the deleted schema
- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
 - Error code 40402 -- Version not found
- [422 Unprocessable Entity](#) --
 - Error code 42202 -- Invalid version
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Status Codes:

Example request:

```
DELETE /subjects/test/versions/1 HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

1
```

Copy

7.6 Compatibility

The compatibility resource allows the user to test schemas for compatibility against specific versions of a subject's schema.

POST /compatibility/subjects/(string: subject)/versions/(versionId: version)

Test input schema against a particular version of a subject's schema for compatibility. Note that the compatibility level applied for the check is the configured compatibility level for the subject (`http:get:: /config/(string: subject)`). If this subject's compatibility level was never changed, then the global compatibility level applies (`http:get:: /config`).

Parameters:

- **subject** (*string*) -- Subject of the schema version against which compatibility is to be tested
- **version** (*versionId*) -- Version of the subject's schema against which compatibility is to be tested. Valid values for versionId are between

[1,2^31-1] or the string "latest". "latest" checks compatibility of the input schema with the last registered schema under the specified subject

Response JSON Object:

- **is_compatible** (*boolean*) -- True, if compatible. False otherwise

Status Codes:

- [404 Not Found](#) --
 - Error code 40401 -- Subject not found
 - Error code 40402 -- Version not found
- [422 Unprocessable Entity](#) --
 - Error code 42201 -- Invalid Avro schema
 - Error code 42202 -- Invalid version
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Example request:

```
POST /compatibility/subjects/test/versions/latest HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

{
  "schema":
  "{
    \"type\": \"record\",
    \"name\": \"test\",
    \"fields\":
    [
      {
        \"type\": \"string\",
        \"name\": \"field1\"
      },
      {
        \"type\": \"int\",
        \"name\": \"field2\"
      }
    ]
  }"
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "is_compatible": true
}
```

Copy

7.7 Config

The config resource allows you to inspect the cluster-level configuration values as well as subject overrides.

PUT /config

Update global compatibility level.

When there are multiple instances of Schema Registry running in the same cluster, the update request will be forwarded to one of the instances designated as the primary. If the primary is not available, the client will get an error code indicating that the forwarding has failed.

Request JSON Object:

- **compatibility** (*string*) -- New global compatibility level. Must be one of BACKWARD, BACKWARD_TRANSITIVE, FORWARD, FORWARD_TRANSITIVE, FULL, FULL_TRANSITIVE, NONE

Status Codes:

- [422 Unprocessable Entity](#) --
 - Error code 42203 -- Invalid compatibility level
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store
 - Error code 50003 -- Error while forwarding the request to the primary

```
PUT /config HTTP/1.1
Host: kafkaproxy.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

{
  "compatibility": "FULL",
}
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "compatibility": "FULL",
}
```

Copy

GET /config

Get global compatibility level.

Request JSON Object:

- **compatibility** (*string*) -- New global compatibility level. Will be one of BACKWARD, BACKWARD_TRANSITIVE, FORWARD, FORWARD_TRANSITIVE, FULL, FULL_TRANSITIVE, NONE

Status Codes:

- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Example request:

```
GET /config HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "compatibilityLevel": "FULL"
}
```

Copy

PUT /config/(string: subject)

Update compatibility level for the specified subject.

Parameters:

- **subject** (*string*) -- Name of the subject

Request JSON Object:

- **compatibility** (*string*) -- New global compatibility level. Must be one of BACKWARD, BACKWARD_TRANSITIVE, FORWARD, FORWARD_TRANSITIVE, FULL, FULL_TRANSITIVE, NONE

Status Codes:

- [422 Unprocessable Entity](#) --
 - Error code 42203 -- Invalid compatibility level
- [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store
 - Error code 50003 -- Error while forwarding the request to the primary

Example request:

```
PUT /config/test HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json

{
  "compatibility": "FULL",
}
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "compatibility": "FULL",
}
```

Copy

GET /config/(string: subject)

Get compatibility level for a subject.

Parameters:

- **subject** (*string*) -- Name of the subject

Request JSON Object:

- **compatibility** (*string*) -- New global compatibility level. Will be one of BACKWARD, BACKWARD_TRANSITIVE, FORWARD, FORWARD_TRANSITIVE, FULL, FULL_TRANSITIVE, NONE
- Status Codes:**
- [404 Not Found](#) -- Subject not found
 - [500 Internal Server Error](#) --
 - Error code 50001 -- Error in the backend data store

Example request:

```
GET /config/test HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json, application/vnd.schemaregistry+json, application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json
```

```
{  
  "compatibilityLevel": "FULL"  
}
```

Please note that all the material included in APPENDIX B is taken from its online source found in [23].

8 APPENDIX C to ANNEX

In this appendix the Kafka Connect API Reference is quoted as it appears in the relative online resource [9] for convenience purposes and as requested:

The material in this appendix is Copyrighted by Confluent, Inc (© Copyright 2019, Confluent, Inc).

Since Kafka Connect is intended to be run as a service, it also supports a REST API for managing connectors. By default this service runs on port `8083`. When executed in distributed mode, the REST API will be the primary interface to the cluster. You can make requests to any cluster member; the REST API automatically forwards requests if required.

Although you can use the standalone mode just by submitting a connector on the command line, it also runs the REST interface. This is useful for getting status information, adding and removing connectors without stopping the process, and more.

Currently the top level resources are `connector` and `connector-plugins`. The sub-resources for `connector` lists configuration settings and tasks and the sub-resource for `connector-plugins` provides configuration validation and recommendation.

Note that if you try to modify, update or delete a resource under `connector` which may require the request to be forwarded to the leader, Connect will return status code 409 while the worker group rebalance is in process as the leader may change during rebalance.

8.1 Content Types

Currently the REST API only supports `application/json` as both the request and response entity content type. Your requests *should* specify the expected content type of the response via the HTTP `Accept` header:

```
Accept: application/json
```

Copy

and *should* specify the content type of the request entity (if one is included) via the `Content-Type` header:

```
Content-Type: application/json
```

Copy

8.2 Statuses & Errors

The REST API will return standards-compliant HTTP statuses. Clients *should* check the HTTP status, especially before attempting to parse and use response entities. Currently the API does not use redirects (statuses in the 300 range), but the use of these codes is reserved for future use so clients should handle them.

When possible, all endpoints will use a standard error message format for all errors (status codes in the 400 or 500 range). For example, a request entity that omits a required field may generate the following response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json
```

```
{
  "error_code": 422,
  "message": "config may not be empty"
}
```

Copy

8.3 Connectors

GET /connectors

Get a list of active connectors

Response JSON Object:

- **connectors** (*array*) -- List of connector names

Example request:

```
GET /connectors HTTP/1.1
Host: connect.example.com
Accept: application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

["my-jdbc-source", "my-hdfs-sink"]
```

Copy

POST /connectors

Create a new connector, returning the current connector info if successful. Return **409 (Conflict)** if rebalance is in process.

Request JSON Object:

- **name** (*string*) -- Name of the connector to create
- **config** (*map*) -- Configuration parameters for the connector. All values should be strings.

Response JSON Object:

- **name** (*string*) -- Name of the created connector
- **config** (*map*) -- Configuration parameters for the connector.
- **tasks** (*array*) -- List of active tasks generated by the connector
- **tasks[i].connector** (*string*) -- The name of the connector the task belongs to
- **tasks[i].task** (*int*) -- Task ID within the connector.

Example request:

```
POST /connectors HTTP/1.1
Host: connect.example.com
Content-Type: application/json
Accept: application/json

{
  "name": "hdfs-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
    "tasks.max": "10",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  }
}
```

Copy

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "name": "hdfs-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
    "tasks.max": "10",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  },
  "tasks": [
    { "connector": "hdfs-sink-connector", "task": 1 },
    { "connector": "hdfs-sink-connector", "task": 2 },
    { "connector": "hdfs-sink-connector", "task": 3 }
  ]
}
```

Copy

GET /connectors/(*string:name*)

Get information about the connector.

Response JSON Object:

- **name** (*string*) -- Name of the created connector
- **config** (*map*) -- Configuration parameters for the connector.
- **tasks** (*array*) -- List of active tasks generated by the connector
- **tasks[i].connector** (*string*) -- The name of the connector the task belongs to
- **tasks[i].task** (*int*) -- Task ID within the connector.

Example request:

```
GET /connectors/hdfs-sink-connector HTTP/1.1
Host: connect.example.com
Accept: application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "hdfs-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
    "tasks.max": "10",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  },
  "tasks": [
    { "connector": "hdfs-sink-connector", "task": 1 },
    { "connector": "hdfs-sink-connector", "task": 2 },
    { "connector": "hdfs-sink-connector", "task": 3 }
  ]
}
```

Copy

GET /connectors/(string:name)/config

Get the configuration for the connector.

Response JSON Object:

- **config** (*map*) -- Configuration parameters for the connector.

Example request:

```
GET /connectors/hdfs-sink-connector/config HTTP/1.1
Host: connect.example.com
Accept: application/json
```

Copy

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
  "tasks.max": "10",
  "topics": "test-topic",
  "hdfs.url": "hdfs://fakehost:9000",
  "hadoop.conf.dir": "/opt/hadoop/conf",
  "hadoop.home": "/opt/hadoop",
  "flush.size": "100",
  "rotate.interval.ms": "1000"
}
```

Copy

PUT /connectors/(string:name)/config

Create a new connector using the given configuration, or update the configuration for an existing connector. Returns information about the connector after the change has been made.

Return **409 (Conflict)** if rebalance is in process.

Request JSON Object:

- **config** (*map*) -- Configuration parameters for the connector. All values should be strings.

Response JSON Object:

- **name** (*string*) -- Name of the created connector
- **config** (*map*) -- Configuration parameters for the connector.
- **tasks** (*array*) -- List of active tasks generated by the connector
- **tasks[i].connector** (*string*) -- The name of the connector the task belongs to
- **tasks[i].task** (*int*) -- Task ID within the connector.

Example request:

```
PUT /connectors/hdfs-sink-connector/config HTTP/1.1
Host: connect.example.com
Accept: application/json

{
  "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
  "tasks.max": "10",
```

```

"topics": "test-topic",
"hdfs.url": "hdfs://fakehost:9000",
"hadoop.conf.dir": "/opt/hadoop/conf",
"hadoop.home": "/opt/hadoop",
"flush.size": "100",
"rotate.interval.ms": "1000"
}

```

Copy

Example response:

```

HTTP/1.1 201 Created
Content-Type: application/json

{
  "name": "hdfs-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
    "tasks.max": "10",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  },
  "tasks": [
    { "connector": "hdfs-sink-connector", "task": 1 },
    { "connector": "hdfs-sink-connector", "task": 2 },
    { "connector": "hdfs-sink-connector", "task": 3 }
  ]
}

```

Copy

Note that in this example the return status indicates that the connector was **Created**. In the case of a configuration update the status would have been **200 OK**.

GET /connectors/(string:name)/status

Get current status of the connector, including whether it is running, failed or paused, which worker it is assigned to, error information if it has failed, and the state of all its tasks.

Response JSON Object:

- **name** (*string*) -- The name of the connector.
- **connector** (*map*) -- The map containing connector status.
- **tasks[i]** (*map*) -- The map containing the task status.

Example request:

```

GET /connectors/hdfs-sink-connector/status HTTP/1.1
Host: connect.example.com

```

Copy

Example response:

```

HTTP/1.1 200 OK

{
  "name": "hdfs-sink-connector",
  "connector": {
    "state": "RUNNING",
    "worker_id": "fakehost:8083"
  },
  "tasks":
  [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "fakehost:8083"
    },
    {
      "id": 1,
      "state": "FAILED",
      "worker_id": "fakehost:8083",
      "trace": "org.apache.kafka.common.errors.RecordTooLargeException\n"
    }
  ]
}

```

Copy

POST /connectors/(string:name)/restart

Restart the connector and its tasks. Return **409 (Conflict)** if rebalance is in process.

Example request:

```

POST /connectors/hdfs-sink-connector/restart HTTP/1.1
Host: connect.example.com

```

Copy

Example response:

```

HTTP/1.1 200 OK

```

Copy

PUT /connectors/(string:name)/pause

Pause the connector and its tasks, which stops message processing until the connector is resumed. This call asynchronous and the tasks will not transition to **PAUSED** state at the same time.

Example request:

```

PUT /connectors/hdfs-sink-connector/pause HTTP/1.1

```

```
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 202 Accepted
```

Copy

PUT /connectors/(string:name)/resume

Resume a paused connector or do nothing if the connector is not paused. This call asynchronous and the tasks will not transition to **RUNNING** state at the same time.

Example request:

```
PUT /connectors/hdfs-sink-connector/resume HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 202 Accepted
```

Copy

DELETE /connectors/(string:name)/

Delete a connector, halting all tasks and deleting its configuration. Return **409 (Conflict)** if rebalance is in process.

Example request:

```
DELETE /connectors/hdfs-sink-connector HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 204 No Content
```

Copy

8.4 Tasks

GET /connectors/(string:name)/tasks

Get a list of tasks currently running for the connector.

Response JSON Object:

- **tasks** (*array*) -- List of active task configs that have been created by the connector
- **tasks[i].id** (*string*) -- The ID of task
- **tasks[i].id.connector** (*string*) -- The name of the connector the task belongs to
- **tasks[i].id.task** (*int*) -- Task ID within the connector.
- **tasks[i].config** (*map*) -- Configuration parameters for the task

Example request:

```
GET /connectors/hdfs-sink-connector/tasks HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 200 OK

[
  {
    "task.class": "io.confluent.connect.hdfs.HdfsSinkTask",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  },
  {
    "task.class": "io.confluent.connect.hdfs.HdfsSinkTask",
    "topics": "test-topic",
    "hdfs.url": "hdfs://fakehost:9000",
    "hadoop.conf.dir": "/opt/hadoop/conf",
    "hadoop.home": "/opt/hadoop",
    "flush.size": "100",
    "rotate.interval.ms": "1000"
  }
]
```

Copy

```
GET /connectors/(string:name)/tasks/(int:taskid)/status
```

Get a task's status.

Example request:

```
GET /connectors/hdfs-sink-connector/tasks/1/status HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 200 OK
{"state": "RUNNING", "id": 1, "worker_id": "192.168.86.101:8083"}
```

Copy

POST /connectors/(string:name)/tasks/(int:taskid)/restart

Restart an individual task.

Example request:

```
POST /connectors/hdfs-sink-connector/tasks/1/restart HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 200 OK
```

Copy

8.5 Connector Plugins

GET /connector-plugins/

Return a list of connector plugins installed in the Kafka Connect cluster. Note that the API only checks for connectors on the worker that handles the request, which means it is possible to see inconsistent results, especially during a rolling upgrade if you add new connector jars.

Response JSON Object:

- **class** (*string*) -- The connector class name.

Example request:

```
GET /connector-plugins/ HTTP/1.1
Host: connect.example.com
```

Copy

Example response:

```
HTTP/1.1 200 OK

[
  {
    "class": "io.confluent.connect.hdfs.HdfsSinkConnector"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSourceConnector"
  }
]
```



```
}
]
```

Copy

PUT /connector-plugins/*(string:name)*/config/validate

Validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

Request JSON Object:

- **config** (*map*) -- Configuration parameters for the connector. All values should be strings.

Response JSON Object:

- **name** (*string*) -- The class name of the connector plugin.
- **error_count** (*int*) -- The total number of errors encountered during configuration validation.
- **groups** (*array*) -- The list of groups used in configuration definitions.
- **configs[i].definition** (*map*) -- The definition for a config in the connector plugin, which includes the name, type, importance, etc.
- **configs[i].value** (*map*) -- The current value for a config, which includes the name, value, recommended values, etc.

Example request:

```
PUT /connector-plugins/FileStreamSinkConnector/config/validate/ HTTP/1.1
Host: connect.example.com
Accept: application/json

{
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
  "tasks.max": "1",
  "topics": "test-topic"
}
```

Copy

Example response:

```
HTTP/1.1 200 OK

{
  "name": "FileStreamSinkConnector",
  "error_count": 1,
  "groups": [
    "Common"
  ],
  "configs": [
    {
      "definition": {
```

```

        "name": "topics",
        "type": "LIST",
        "required": false,
        "default_value": "",
        "importance": "HIGH",
        "documentation": "",
        "group": "Common",
        "width": "LONG",
        "display_name": "Topics",
        "dependents": [],
        "order": 4
    },
    "value": {
        "name": "topics",
        "value": "test-topic",
        "recommended_values": [],
        "errors": [],
        "visible": true
    }
},
{
    "definition": {
        "name": "file",
        "type": "STRING",
        "required": true,
        "default_value": "",
        "importance": "HIGH",
        "documentation": "Destination filename.",
        "group": null,
        "width": "NONE",
        "display_name": "file",
        "dependents": [],
        "order": -1
    },
    "value": {
        "name": "file",
        "value": null,
        "recommended_values": [],
        "errors": [
            "Missing required configuration \"file\" which has no default value."
        ],
        "visible": true
    }
},
{
    "definition": {
        "name": "name",
        "type": "STRING",
        "required": true,
        "default_value": "",
        "importance": "HIGH",
        "documentation": "Globally unique name to use for this connector.",
        "group": "Common",
        "width": "MEDIUM",
        "display_name": "Connector name",
        "dependents": [],
        "order": 1
    },
    "value": {
        "name": "name",
        "value": "test",
        "recommended_values": [],
        "errors": [],
        "visible": true
    }
}

```

```

    },
    {
      "definition": {
        "name": "tasks.max",
        "type": "INT",
        "required": false,
        "default_value": "1",
        "importance": "HIGH",
        "documentation": "Maximum number of tasks to use for this connector.",
        "group": "Common",
        "width": "SHORT",
        "display_name": "Tasks max",
        "dependents": [],
        "order": 3
      },
      "value": {
        "name": "tasks.max",
        "value": "1",
        "recommended_values": [],
        "errors": [],
        "visible": true
      }
    },
    {
      "definition": {
        "name": "connector.class",
        "type": "STRING",
        "required": true,
        "default_value": "",
        "importance": "HIGH",
        "documentation": "Name or alias of the class for this connector. Must be a subclass of org.apache.kafka.connect.connector.Connector. If the connector is org.apache.kafka.connect.file.FileStreamSinkConnector, you can either specify this full name, or use \"FileStreamSink\" or \"FileStreamSinkConnector\" to make the configuration a bit shorter",
        "group": "Common",
        "width": "LONG",
        "display_name": "Connector class",
        "dependents": [],
        "order": 2
      },
      "value": {
        "name": "connector.class",
        "value": "org.apache.kafka.connect.file.FileStreamSinkConnector",
        "recommended_values": [],
        "errors": [],
        "visible": true
      }
    }
  ]
}

```

Please note that all the material included in APPENDIX C is taken from its online source found in [9].

9 APPENDIX D to ANNEX

In this appendix high resolution images of the fully detailed class diagrams are made available for reference purposes. **The images are of sufficient quality and can be zoomed in for readability.**

9.1 Kafka Producer type Connector for the VFI historical data Class Diagram

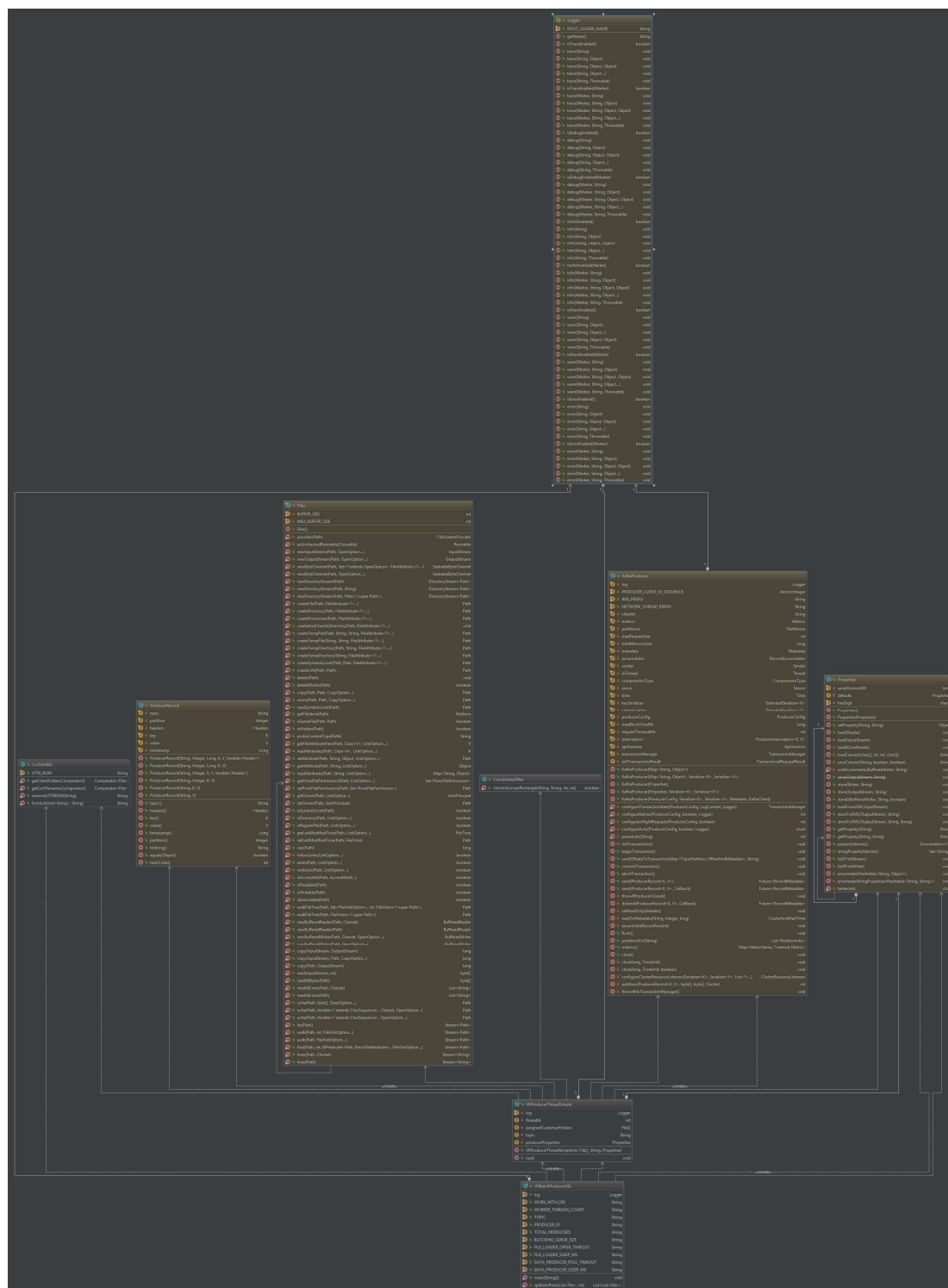
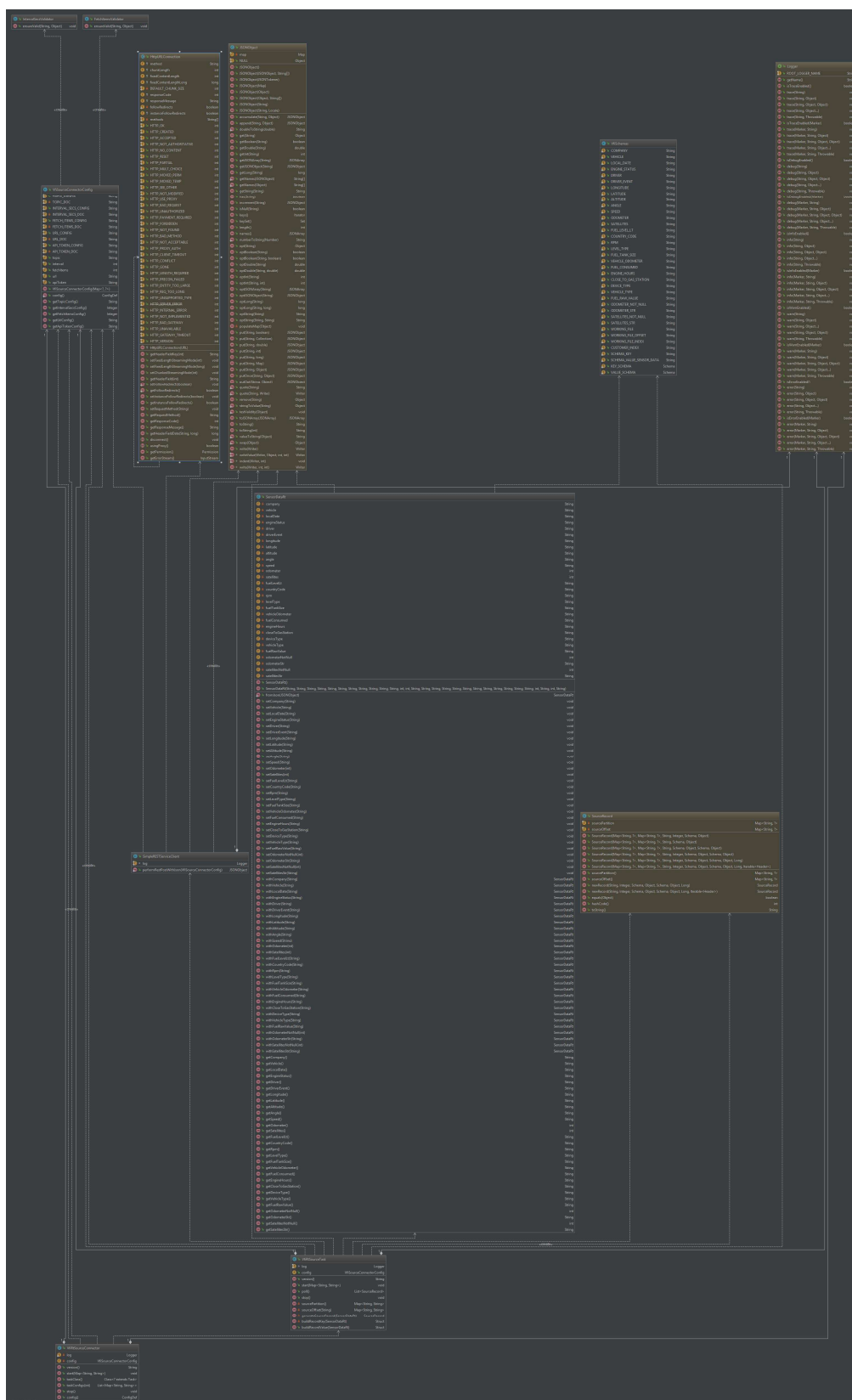


Figure 9.1 Kafka Producer type Connector for the VFI historical data Class Diagram

Page | 200



9.3 Kafka Connect type Connector for the SIS METRICS data Class Diagram

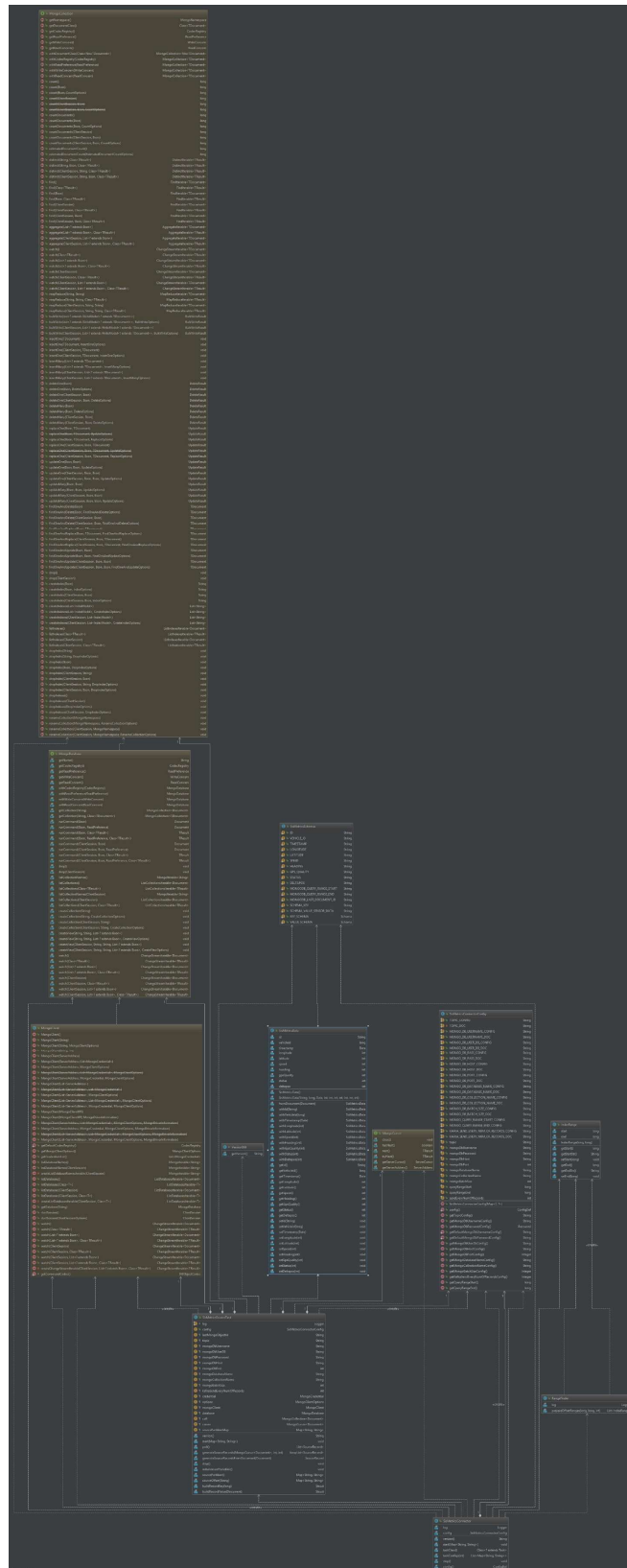
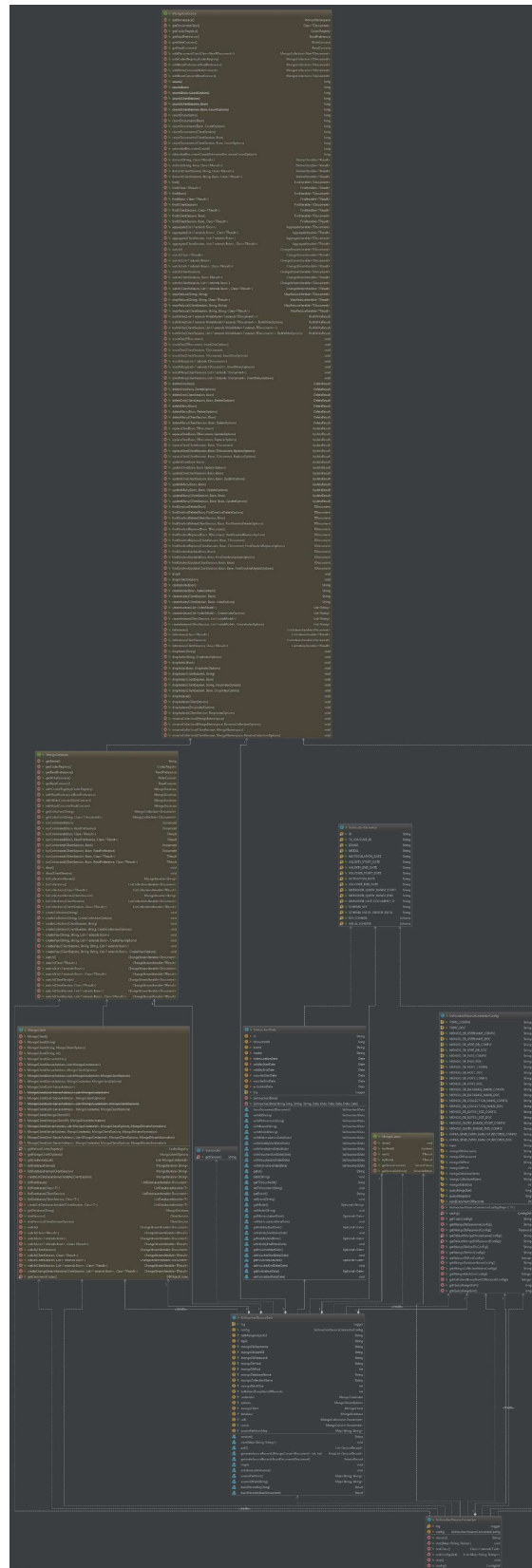


Figure 9.3 Kafka Connect type Connector for the SIS METRICS data Class Diagram

Page | 202



9.5 Kafka Producer type Connector for the SIS METRICS data Class Diagram



Figure 9.5 Kafka Producer type Connector for the SIS METRICS data Class Diagram

9.6 Kafka Producer type Connector for the SIS VOUCHER data Class Diagram

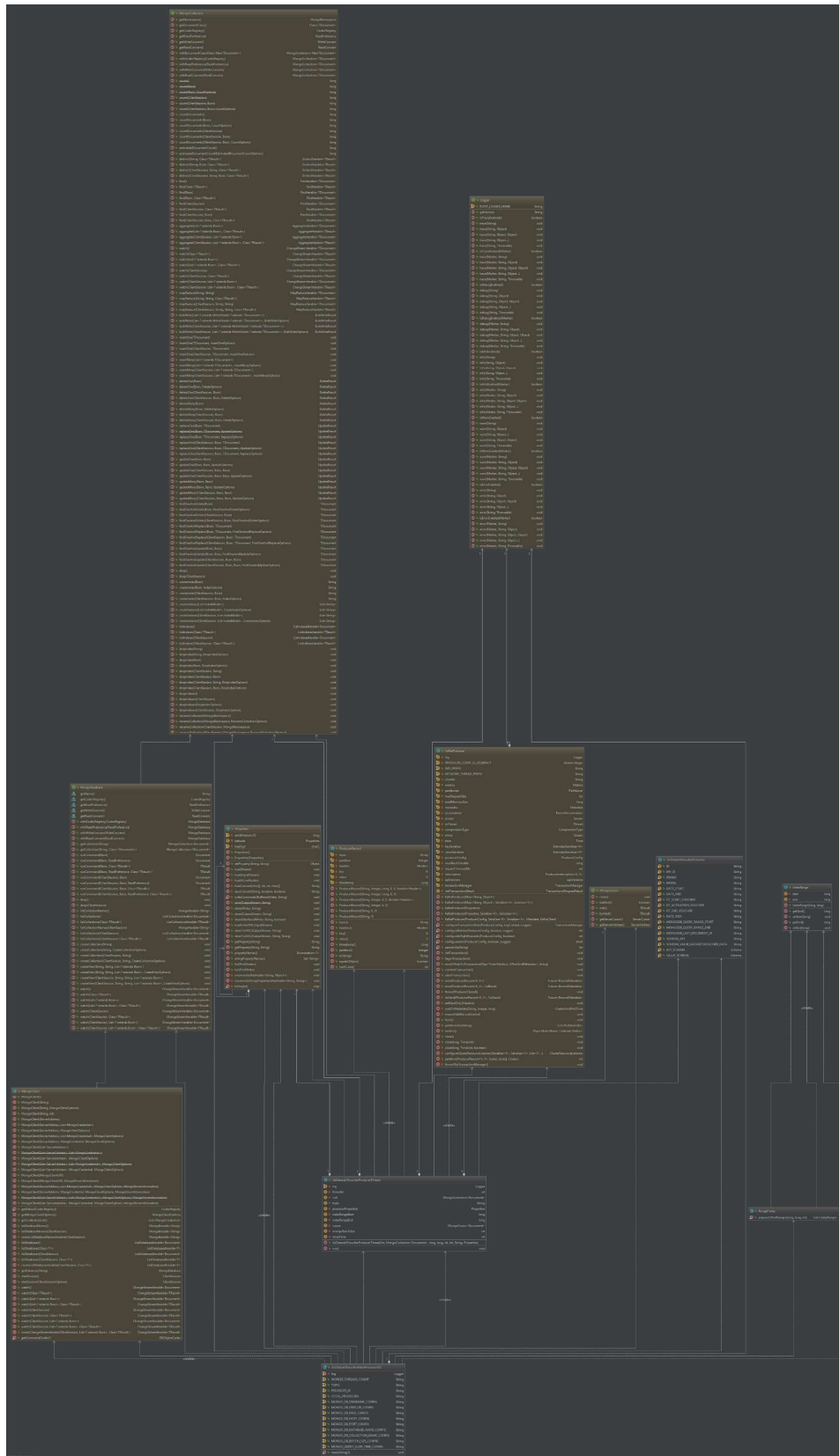


Figure 9.6 Kafka Producer type Connector for the SIS VOUCHER data Class Diagram



Page | 206

