Projectdocumentation for Open Environemental Data Cloud Prototyping Project

Nico Tasche, ...

Technische Universitaet Berlin, Strae des 17. Juni 135, 10623 Berlin, Germany,
nico.tasche@campus.tu-berlin.de

Abstract. **Keywords:** Open Data, Environemental Data, Big Data

1 Introduction

Authorship

Version	Date	Modified by	Summary of changes
0.1.0	2017-07-19	Oliver, Andres	Outline & working draft
0.1.1	2017-07-20	Paul	Motivation section first draft
0.2.0	2017-07-22	Oliver, Andres	Motivation rewrite & flesh out Requirements section
0.2.1	2017-07-29	Oliver, Andres	Fixes for Markdown-> LaTeX conversion
0.2.2	2017-07-29	Oliver	Removing content about Organization (moved to new document)

1.1 Summary/Overview/Abstract

1.2 What is Open Data

So what do we mean when we talk about “open data”?

> Open means anyone can freely access, use, modify, and share for any purpose (subject, at most, to requirements that preserve provenance and openness). http://opendefinition.org/

Further, the definition encompasses the following aspects:

– **Availability and Access**: the data must be available as a whole and at no more than a reasonable reproduction cost, preferably by downloading over the Internet. The data must also be available in a convenient and modifiable form.
– **Re-use and Redistribution**: the data must be provided under terms that permit re-use and redistribution including the intermixing with other datasets.
– **Universal Participation**: everyone must be able to use, re-use and redistribute - there should be no discrimination against fields of endeavour or against persons or groups. For example, 'non-commercial' restrictions that would prevent 'commercial' use, or restrictions of use for certain purposes (e.g. only in education), are not allowed. http://opendatahandbook.org/guide/en/what-is-open-data/

Time series data & non-static sensor environmental data So how does this "open data" differ from the data we're interested in? The simple answer is the lack of the time dimension in existing data. The large majority of open data out there today is static. Environmental data are usually represented on discrete "datasets", but there is no connection between data of the same source or type which was collected at different times. For example, one might find a dataset about a particular environmental measurement such as average and maximum water pollution on a given set of geographical areas or points. However, the same data for the following month or year are published separately, with no connection to the first, and at times having different formats or semantics.

We therefore distinguish between static data of the type described above (of which there is an abundance) to the data which are in scope for the project, namely time series data of environmental measurements coming from a device (i.e. a sensor or sensor network) with some form of geo-information. For example, an array of air pollution sensors in a given city may collect data every at 15-minute intervals; this would be represented as individual records containing the timestamp, the sensor's geolocation and the air pollution measurement.

1.3 Motivation

Open data efforts in the past have focused on providing a centralized platform onto which data producers can upload their data along with some metadata but without much (if any) concern for the schema or the format in which the data is offered. This has resulted in very large independent and heterogeneous catalogs of data which are difficult to discover and integrate without significant manual effort.

The advent of the Internet of Things (IoT) has also meant that unprecedented amounts of data are generated by millions of devices every second of every day. However, devices generates data in their own (often closed-source proprietary) format, thanks to a lack of a common or widely established data model for environmental data. This results in lots of data from which it's difficult to gain insights due to the inherent difficulty in querying data in disparate representations.

Also, as the price of devices declines, more and more enthusiasts are willing to share their data to open communities so that it can be used and queried by anyone. To tackle these challenges our project centers around building a prototype which provides:

– a platform on which data owners can share their sensor-generated environmental data,
– a unified schema to support queries across different data from heterogeneous sources
– a simple and extensible framework to facilitate the data import process,
– and a flexible querying interface for accessing the data.

With this we aim to create a tool that generates economic and social value through new and creative "layering" of data.

`//TODO` there is very little background to talk about specific stuff in the req's (like units & measurements if those concepts haven't been introduced. Move the decomposition & req's further down & pull up the open-data & other introductory text about our domain here)

Decomposition The problem domain can be decomposed into the following components:

1. Data Import Framework
2. Database
3. Public API
4. Cloud Infrastructure

1.4 Requirements

The above objectives translate into the following requirements:

General

1. **Open Source Software**: Libraries & components used shall be open source software.
2. **Cloud architectural style**: Guiding architectural principle shall be to avoid monolith-style applications, but rather include cloud concerns from early on (i.e. design phase).
3. **Scalability**: Components shall be inherently scalable.

4. **Fault tolerance**: Components shall provide fault tolerance capabilities.
5. **Performance**: The system (and its constituent components) shall have the ability to handle extremely high demand.
6. **Portability**: The system shall be deployable both on public and private clouds.

Data Import Framework

1. Provide facilities for common access patterns of data sources (e.g. FTP, HTTP)
2. Provide facilities to read data in common formats (e.g. JSON, CSV, XML)
3. Provide facilities to map the user schema to the platforms common schema
4. Provide reusable community-based unit converters

Database

1. Ability to perform time series and geolocation range queries

Public API `TODO` prob just the Scalability & fault tolerance concerns already mentioned under General, so not sure what reqs we had for the API individually (if any)...

1. RESTful
   – Stateless

2 Competitor Analysis

Authorship

Version	Date	Modified by	Summary of changes
0.1	2017-06-03	Andres Ardila	Working draft
0.2	2017-07-07	Andres Ardila	Added secion on Talend

```
//TODO diagram illustrating the pipeline to better describe where each company is positione
1.
```

2.1 OpenSensors.io

https://www.opensensors.io

Features

> Securely manage your private IoT network [...] whether you are a startup or an enterprise

– IoT device management
– Real-time and historical APIs
– Analytics
– Ability to run infrastructure in user's own dedicated cloud network
– Integration with user's backend systems
– Secure data sent to the cloud using TLS
– Set policies on who can sees the data by users or groups of engineers
– Triggers when data hits certain thresholds and alert engineers when device is down

Hardware agnostic Provides support for open protocols, use existing SDKs to quickly get up and running using MQTT or HTTPS protocols. Their partnership program includes hardware providers and manufacturing firms.

Segments

– **Workspace planning**: "Use sensors and our data platform to understand if you are using it efficiently and forecast your future needs."
– **Environment sensing**: Open Data communities to contribute and use data from environmental sensors around you. You can get air quality data and traffic data for free.

Customers

– **Customers** (TODO)
– **Partners** are hardware manufacturers and design firms (OEMs) looking to deliver client projects

Analysis

– Not really open. There is no sign-up with which one can get access to the allegedly "open" data. As a result, one can conclude that their focus is the OEM sector as opposed to "data scientists looking to create interesting mash-ups of data."
– . . .

2.2 PubNub

https://www.pubnub.com/

Features

Scalability 15 globally replicated points of presence transacting over 1.5 trillion messages for 300 million unique devices per month. #### Storage 15 globally replicated points of presence transacting over 1.5 trillion messages for 300 million unique devices per month. ### Segments ### Customers ### Analysis

2.3 Talend

https://www.talend.com/resource/sensor-data/
Talend

Features

Segments

Customers

Analysis

3 Architecutre

Authoriship

Version	Date	Modified by	Summary of changes
0.1	2017-07-19	Oliver, Amer	Working draft
0.2	2017-07-28	Oliver	Tidy up . . .

- Components Overview/Description (if applicable ***Motivation***)
- Requirements (specific to this component)
- Survey of Existing Solutions (available implementations)
- Evaluation Criteria & Decision-making Process
- Implementation Details
- Evolution of Component during development (Reasons for the Changes)
- Critical Analysis/Limitations
- Future Development/Enhancements

3.1 Requirements of the whole platform

- scalability
- define scalability/ our interpretation of this requirement specifically for our platform
- extensibility
- define it!
- objective was to create a whole pipeline
- Collection data from various sources
- Process/transform the data
- Store the data persistently
- Provide the data through a "single" interface
- no usage of cloud/provider specific solutions

3.2 Design decisions

To tackle all of the mentioned requirements, we decided upon the following architecture [image of architecture]. - We traded consistency for availability. (Harvest over yield {{ref}}) - "Decentralized system": To be able to scale the platform horizontally, it has to be distributed. Thus, smaller pieces of the system have to work on different/separate machines. We omitted bottlenecks by picking components that are distributed by design.

Importers

– Every source is different! Heterogeneous interfaces, different data formats, different protocols
– Sources have specific limitations, e.g. number of requests per specific time slot
– Size and frequency of data points is heterogeneous
– Availability of sources is different –> Every importer runs as a independent service. All have specific lifecycles (run-time, frequency of execution). Possibility to schedule them independently. –> Upon finishing the importing task, the running container is terminated to allow for maximal resource utilization. –> Every importer manages its state independently from the system. If the importer failed completing the importing task, it continues from the last checkpoint. –> Every importer has a unique name when scheduled to avoid repetition of the same importing job.

Messaging System

– Messaging enabled the component to decouple the services from each other. We used queue systems to transport and buffer messages to decouple the components even further.
– Used a distributed messaging systems that is capable of handling millions of concurrent requests and is fault-tolerant.
– Autonomous parts can be deployed independently, such that the platform keeps running without interruption in contrast to deploying a monolithic application.
– Choice of open source solutions to ease portability and make it cross-platform

3.3 Evolution of the architecture

– Filebeat got obsolete
– Scheduling handeled by Kubernetes
– Validation and insert into one component
–

3.4 Limitations

– ???

3.5 Future

- Connection from public API to relational system
-

4 Extensible Data Import Framework

Authoriship

Version	Date	Modified by	Summary of changes
0.1	2017-07-20	Rohullah, Jawid	initial version
0.1a	2017-07-21	Rohullah, Jawid	framework description
0.1b	2017-07-22	Rohullah, Jawid	evaluations, features, improvements, values parts
0.2	2017-07-26	Andres	Proofread: spelling, readability, etc.
0.2a	2017-07-28	Andres	Further refinement of text
0.3	2017-07-28	Rohullah, Jawid	framework structure explanations + diagrams

The Open Data Platform which we have built for extracting, transforming and loading open sensor data is made up several significant components to efficiently process the large quantity of incoming data.

The first part of the system requires a powerful tool to do the job of importing very large amounts of different kinds of data with various formats and types from several data sources in a performant manner.

The job of the importers, in addition to fetching data from its source, is mainly to apply a series of rules and transformations to the source data in order to fit the our schema and be stored in the system.

For this purpose, a framework that supports the entire processes of extracting and transforming data is desirable.

In addition to the main import functionality, additional requirements and criteria were considered regarding the framework.

This chapter describes the requirements, selection criteria, design decisions, evaluations, technical implementation details, further possible improvements, as well as the value of having such a framework.

4.1 Framework Requirements

The following are use-cases which our framework should cover:

- user can add a new data source with minimal coding and configuration effort.
- provide various functionalities to the user such as processing data into customized format/schema
- reusable components for reading, processing and writing data.
- ideally the framework may be provided as a starter application which doesn't require it be built from scratch every time.
- the framework should support extracting data from known transport channels and in known formats.
- the entire framework must be based on microservices architecture (as opposed to a monolithic system).
- the framework should include logging functionalities.
- if an importer were to crash while importing data, it should continue from the point where it failed after being restarted.

4.2 Evaluation of different data import frameworks

According to the framework requirements, we needed to search and find a useful and powerful tool such as an existing ETL framework or a technology on top of which we could build our data import framework. Therefore, it seemed to be a good idea to compare these frameworks and come up with the best decision. So we compared a couple of existing frameworks and technologies with different aspects such as:

- Functional and non-functional aspects of different frameworks are considered.
- Popularity of its programming language
- Popularity of its user community
- Open source or open license
- Capable of being deployed as a microservice to the Cloud
- Capable of scheduling the jobs
- Processing of jobs into batches
- Recoverability of individual jobs from failures

Evaluated Existing Frameworks

1. Spring Batch

  - Spring Cloud Tasks
  - Spring Cloud Data Flow
2. Java EE
3. Easy Batch
4. Summer Batch
5. Talend ETL

4.3 Design Decisions

After evaluation and comparison, it was decided to implement our extensible framework using **Spring Batch** given the features and functionality it offers.

Why Spring Batch

- Lightweight, ready-to-use framework for robust batch processing
- Suitable framework for data integration and processing
- Popular with a large community of users
- Written in a popular language (Java)
- Its *Cloud Task* feature allows the deployment of data importers as microservices
- Capabilities for scheduling the jobs in data processing pipeline
- Familiarity to some team members

Spring Batch Features

1. Reusable architecture framework
2. Lightweight, enterprise and batch job processing
3. Open Source
4. Reusable functions such as:
   - logging/tracing
   - job processing statistics
   - job restart
   - transaction management
5. Concurrent batch processing: parallel processing of a job
6. Manual or scheduled restart after failure
7. Deployment model, with the architecture JARs, built using Maven.
8. The ability to stop/start/restart jobs and maintain state between executions.

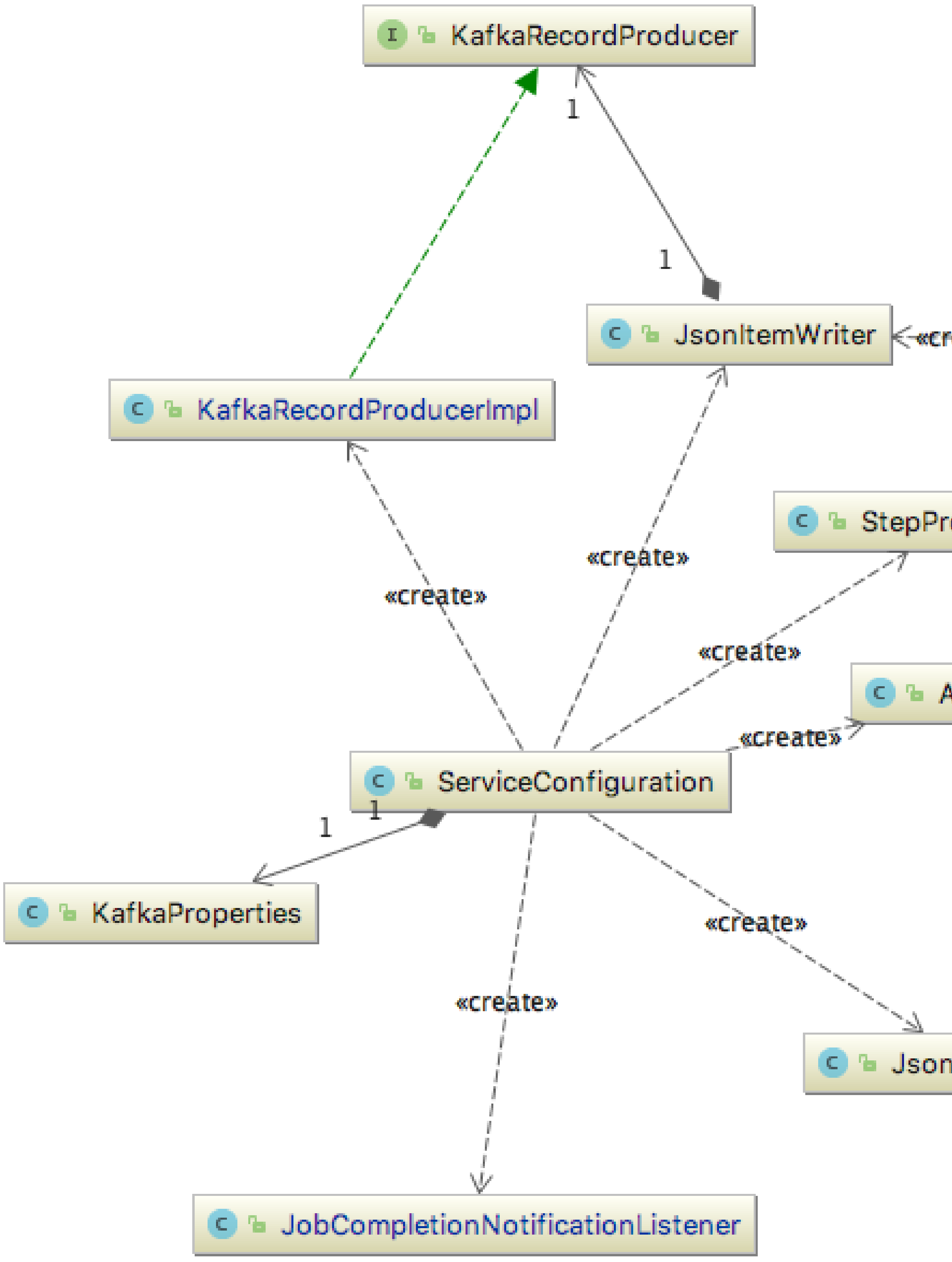
4.4 Framework Structure/ Architecture

Figure 1. shows an abstract overview of a simple importer application. We will look at each component individually.

- **ImporterApplication:** Entry point of the importer (Java main class).

Fig. 1. image-title-here

– **BatchConfiguration:** Includes configurations on how to read, process and write items. It also includes listeners, batch jobs and job steps.
– **Listeners:** Are defined for tracking jobs and steps and logging the process of importing to the console.
– **Jobs:** Generally there is one batch job for each importer. A batch job may have one or more steps for importing a single source. - **Steps:** Each step includes a reader, processor, and a writer. - **Reader:** Defines how to read items from the source. - **Processor:** Processes every item - an item is basically an object that represents an record - individually and creates a JSON string for that according to our defined schema
- **Writer:** Simply writes to Kafka queue.
– `@SpringBootApplication`: Annotation to make the application a Spring Boot Application
– `@EnableBatchProcessing`: Annotation to add the functionality of processing batch jobs.
– `@EnableTask`: Enables the deployment of data importers as microservices which shutdown once importing is finished.
– **ServiceConfiguration:** Is the component where the services are registered as Java Beans for re-usability. It is included in module 'library' where reusable classes across all importers are defined. The following main components are included in the 'library' module. - **ServiceConfiguration** - **ApplicationService:** Includes some generally used methods to bring facility to importing. - **JsonItemWriter:** Asks `KafkaRecordProducer` to write individual JSON objects to Kafka queue - **KafkaRecordProducer:** Is the class where the items are written to Kafka queue


KafkaRecordProducer
1
1
JsonItemWriter
«cr
KafkaRecordProducerImpl
StepPr
«create»
«create»
«create»
A
«create»
ServiceConfiguration
1
1
KafkaProperties
«create»
«create»
Json
JobCompletionNotificationListener

4.5 Framework Features

- Ability to import data with various types and formats
- A Module with pre-packaged utility classes
  - Just import and ready to use utility classes
- Independent Cloud Tasks as microservices
- Logging and tracing execution of jobs in different steps

4.6 Framework Strengths

- Usability
  - Reusable, ready-to-use functionalities
- Extensibility
  - Ability to add custom utilities
  - Easy to add new jobs
- Portability
  - Jobs run as microservices
  - Every importer could be packed into JAR file and deployed into private or public cloud

4.7 Supported Data Formats

The sources which we used have come in various types and formats. Therefore different data formats and types have been implemented for importing through our framework such as: - REST Interface, HTTP and FTP data source types - Delimiter-separated value (DSV) - CSV - TSV - HTML - XLS - XLXS

4.8 Possible Further Improvements

1. Scheduling jobs for every importer within the Framework: until now, the scheduling of importing jobs is done by Kubernetes, but this functionality could be provided by Spring via an embeddable component such as (Quartz). This feature would easily allow the scheduling of jobs at specific times.
2. Recoverability of jobs from failures: Currently the intermediate results of job processing are stored in an in-memory database (H2) inside the importer. As we containerized every importer into Docker containers, this functionality will disappear when the container terminates. The solution would be to create and configure a single relational database to store all the intermediary results of the jobs executions; whenever an importer crashes and is restarted, it will start from the last point that it stopped.

4.9 The Values of the Framework

- The application of our already built framework is mostly required for huge data integration and migration.
- Useful functionalities are ready to use just by importing them.
- It is very easy to extend it by adding new data sources to import and some new functions such as unit conversions.
- Writing transformed data to the pipeline (i.e. the queue) is abstracted away from the user and provided by the framework.
- Well defined, clean code with clear comments.

5 Inserting into Elasticsearch

Authoriship

Version	Date	Modified by	Summary of changes
0.1	2017-07-24	Paul	initial version

This section is about our consumer component, that takes data items from a queue and inserts them into the main database. Why we decided to do so and what tasks it fulfills will be discussed.

5.1 Requirements

With our architectural design importers push their results to a queue, in our case *Apache Kafka*. From the queue the data items still must be inserted into Elasticsearch. Besides simply taking data from the queue there are some more requirements. Those mainly are:

Validate input before inserting While designing the ETL-importing pipeline we had the problem, that the output of data-fragments (JSON documents) that are prepared and processed for insertion must be validated previously to inserting. That task is logically linked to the data storage component and not the importing process itself as

- the ETL-framework should not validate its own outcome (as it or its creator are biased)
- the database should asure, that it is not importing corrupted data-fragments that will break queries etc.

The then logical place to do the validation was the component that first processes the data items after the importers released them.

Ability to scale and adjust the configuration according to Kafka's configuration Kafka can be configured in many ways. This mainly is about distribution (how many brokers), partitioning and the topics (chanells) it uses. The consumer should work and be reconfigurable, if our kafka configuration chances

Allow parallelism for reading from Kafka Depending on the configuration there can be several options, how parallel reading from Kafka can be accomplished (consumer groups, many topics and consumers, etc.). The consumer should be able to work for all of them or at least the ones we decide for.

Achieve high performance to not slow down the importing pipeline As our importers reached very high speeds in producing data items ($>$ 1 million in few seconds) and many of them can be run in parallel the consumer itself shell also deliver a good performance. While of course overall we want to tackle every possible bottleneck, for the consumer itself it was at first most important that it is not the bottleneck itself.

Allow concurency for processing data read from Kafka A naive strategy of serially reading from the queue, processing the data and the pushing it to Elasticsearch will most likely be way to slow. Therefore we must go for a more sophisticated method, that uses asynchronisity and concurrency to manage a non-blocking higher performant process.

5.2 Implementation of the Requirements

Because of the named reasons we decided to build one component to validate the output and push it - if valid - to Elasticsearch. There would have been the possibility to do this in seperate steps with another queue inbetween aswell. This would have meant a lot higher configurational effort and more resources required so that it seemed good to join both tasks in one component.

Bulk insertion The most important decision was on how to handle the insertion to Elasticsearch. These requests require an http-connection. If we open and close one http-connection per item inserted, the overhead would be enormous. Therefore we wanted to use Elasticsearch's bulk-insertion feature. The http-requests should aswell be non-blocking and be processed in the background to not halt the execution of the other important tasks, like listening to the queue and validation the data.

Achieve high performance and allow validating In order to fulfill the requirements we went with *Go* as a programming language. The main reasons for that was the outstanding performance and the build in concurrency features, which directly tackled the main requirements. Parallelism, configuration and scalability was achieved independent from the programming language and the consumer itself by deploying it the right way. This will be described in the paragraphs after this about go-specific advantages.

Although parallelism can be achieved by simply spawning more consumers (see later) it was still very important to maintain a very fast performance within one importer. While reading from a queue, validating and inserting seemed like a typically task for a scripting language, the performance shortfalls of interpreted languages are too high to use one for this time-sensitive task. Needless to say we also discovered time issues with some test-consumers written e.g. in *ruby*.

For validating the json-items there is the need for a library that handles validation of the used *jsonschema*-format. There is at least one library present for the most relevant programming languages, so this did not limit us in chosing a programming language.

Allow concurency for processing data read from Kafka This was another main reason to chose *go* as a programming language as concurrency is very well supported by the build-in go-routines. With this we could achieve a procedure that works like this:

- Continuously listen to the Kafka queue for new item on the topic.
- Directly validating them with a preloaded schema that is already in memory and therefore takes nearly no time
- Asynchronously passing the validated json to another go-routine, that aggregates the items to bulks until
  - The bulk limit is reached or
  - A timeout is triggered (from outside the routine, needed if importing is very slow, or especially for the last items released by the importer)
- the aggregated json is again asynchronously sent to another go-routine, that handles the http connection to Elasticsearch in the background.

Allow parallelism, scaling and configuration Scaling and configuring the consumer would have been possible in any language: scaling is a job solved by the deployment on a kubernetes cluster and configuring it can be done with environment variables and the same deployment component setting them accordingly. Therefore the parallelism required was also not a requirement directly to the consumer itself.

There are two possibilities to allow parallelized reading from kafka for a data source:

1. Using Kafka's Consumer Groups to allow several consumers reading from one topic
2. Using a topic per importer instance and not per datasource (e.g. when several importers run for several days for a datasource.)

Both of them have been tested by us and both of them have their advantages. As Elasticsearch was the limiting component in an end-to-end importing pipeline by not being able to insert > 5000 records/second with our configuration, we have not been able to do a proper benchmark on how consumer groups behave performance-wise and if it can reach the same performance than using a consumer per importing instance.

We could not achieve to run an even bigger configuration of Elasticsearch. The consumer therefore supplied a speed that reaches Elasticsearches limits as soon as two of them are run in parallel, which can be evaluated as a succes and reduces the need of a detailed benchmark, which options would be more effective.

The consumer is able to join a consumer-group, which can be configured by environment variables and therefore by the deployment-component. If a datasource needs several consumers for one importer (e.g. having lots of data for an importing interval - which is typically a day - that is produced faster than a consumer can utilize) several of them can be spawned inside one consumer group for that topic. If one importing interval can be processed by one importer but there are several of them run in parallel we can create a topic per importing instance and run a consumer for each. So the consumer in its current state is capable for both options.

5.3 Benchmarks

6 Database

Authoriship

Version	Date	Modified by	Summary of changes
0.2	2017-07-28	Tasche, Nico	changed to new structure
0.1	2017-07-24	Tasche, Nico	first working draft

6.1 Overview

6.2 Requirements

Requirements

- scale-able in the range of petabyte in size
- hundred of thousands of requests per minute
- high availablility
- partition tolerance
- fast to handle timeseries
- fast to handle geolocation data
- immediate consistency is NOT necessary

As seen in the requirements, a hugh focus was on scalability. We had some secondary reqirements as well, which were mainly regarding our possibilities to handle the project. ### Requirements - Open source or at the very least an open license is required. - must be well documented - must be managable regarding administration and learning afford

6.3 Survey of Existing Solutions

6.4 Evaluation Criteria & Decision-making Process

The process of deciding what database architecture to use we started with our requirements.

Espacialy the last point of our secondary requiremnts had to be taken into account, because we had no real database expert in our team, so we first considert database-system we allready knew. Our approach was to check whether those databases fulfill our requirements first.

Any relational database as main datastorage has been quickly disreagarded, cause of the bad fairly scaling behavir with the amount of data we have to handly. ## Implementation Details ### Intro to Elasticsearch Elasticsearch is an opensource Lucene based search engine. It is under active development, with an extensive documentation. ### Architecture Each index can be sharded and each shard can have multiple indieces.

TODO: Picture of architecture

To better distribute search requests, the workload is divided among all shards belonging to an index. Because that would not scale very well and would have no partition tolerance, each shard has a configurable number of replicas. A new search request is send to on replica of each shard.

Data model We decided to have an data model which is data-source-centric with the extra posibility to partition the data over time. That means, each data source gets it own index with its own timeframe and its own adjusted datastructure. All our data sources save a few basic data point with each element stored in the database, in particular are those: - timestamp: when has the datapoint been recorded - location: where has the datapoint been recorded

Those are acutally the only information we need to store, besides the individual measurements. We do acutally store some more information, but regarding the common usecases for searches those two datapoints are enough for environemental data. Please refer to the full data-model in the appedix for more information.

This data-model has multiple advantages: - it keeps the data provenance - it allows us to adjust the server infrastructe based on the data source - it scales indefinitely - index size is deterministic, cause of time based partitioning

So why does it scale so good? When importing data from one source, I process and store the data points in one index. This index is not just limited to the data source, it is also limited to the time, e.g. 2016. That means, when 2016 is finished with importing data, the index is done and can be closed up, no one needs to care about it anymore. After the index is done, it might even be transfered to another Elasticsearch node with different hardware. That would be usefull, for example, when the average density of the smurf population is beeing stored. The index can be transferd to a less powerfull hardware with fewer CPU cores and spinning harddrives and even fewer replicas, because this information is probably hardly requested.

Query optimization Why do we need query optimization? For that I'm going to give a small small example to consider: 1. we import multiple sources, with multiple messurements: source1(airtemperature, watertemperatur) 1980-2017, source2(airtemperature) 1983-1990, source3(uv-index) 2009-2017 2. each source is partitioned by year and source 1 is partitioned by month for all data after 2015. 3. every index is naivly sharded over 3 nodes

Let's make a simple search request: give me all uv values data from 2015 till 2017 and aggregate an everage over the month. Because the user does not now anything about the internal database architecture (at least he should not) he requests the temperature and the timeframe.

Worst case: A search request in send to all indieces, that means:

```
source1 = 35 years + (2years x 12 month) x 3 shards
source1 = 177 shards

source2 = 17 years x 3 shars
source2 = 51 shards

source3 = 8 years x 3 shars
source3 = 8 years x 3 shars

source1 + source2 + source3 = 252 shards
```

So in worst case each shard has its own node(very unlikely), the search request has to be send to 252 nodes/computers.

First optimization, Limit the time With a naive approach by checking the common time part of the request 2016-2017 and limit the indieces search with the following pattern:

```
indexsearch: *-201*

source1 = 5 years + (2years x 12 month) x 3 shards
source1 = 87 shards

source2 = 0 years x 3 shars
source2 = 0 shards

source3 = 3 years x 3 shars
source3 = 9 shards

source1 + source2 + source3 = 96 shards
```

We allready reduced the number of shards we need to address by 61%

With a little more sophisticated timelimitation algorithm, we could acctually do more and search just those two years:

```
indexsearch: *-2016, *-2017

source1 = 1 years + (2years x 12 month) x 3 shards
source1 = 75 shards

source2 = 0 shards

source3 = 2 years x 3 shars
source3 = 6 shards

source1 + source2 + source3 = 81 shards
```

Now we are at 68% reduction.

Second optimization, Limit to indieces which contain the right data If we store in a seperate database, which data source and therefore indiece actually holds the requested data we can do even much more:

```
indexsearch: source3-2016, source3-2017

source1 = 0
source2 = 0 shards
source3 = 2 years x 3 shars = 6 shards

source1 + source2 + source3 = 6 shards
```

By using those two optimitzations, we were able to reduce the number of requeseted shard to 6, which means a total reduction of 96.2%.

This was just a naive example. In reality the reduction should even be much higher, with a growing number of data sources. Let's say we have allready 100 data sources and we can limit a request to just two of those for example because the requested messuremnt it provided just by those two, the saving of network traffic and workload would be immense.

6.5 Critical Analysis/Limitations

Joins One mayor drawback of elasticsearch is the missing possibility of server side join, the way they are known by SQL based database-system. This means, any kind of join operation has to be done either on a seperate server, like our api instance, or on the application side. This is actually something we were not really aware of for a long time.

6.6 Future Development and Enhancements

7 Data Source Metadata Management System

Authoriship Written by Paul Wille *Proofread & edited by Andres*

We decided to have a separate component to manage the metadata about data sources which are imported to the system. This component shall fulfill several tasks that are important for organizing the ETL process. Furthermore, it provides important information to several other system components.

In this chapter we will discuss why we decided to include such a component, what it was built for, and how it was realized and implemented.

7.1 Purpose

The main purpose of the component is to:

- serve as a registry of data sources in the system,
- provide data source metadata to other components in the system,
- provide users with information about units of measurement,
- enable users who want to use the data to see the resources our system contains

Data Sources Registry Before data from a given source can be imported, important information about the source must be made available to our system. This information consists of static metainformation and has no direct relation to actual data being provided in the source.

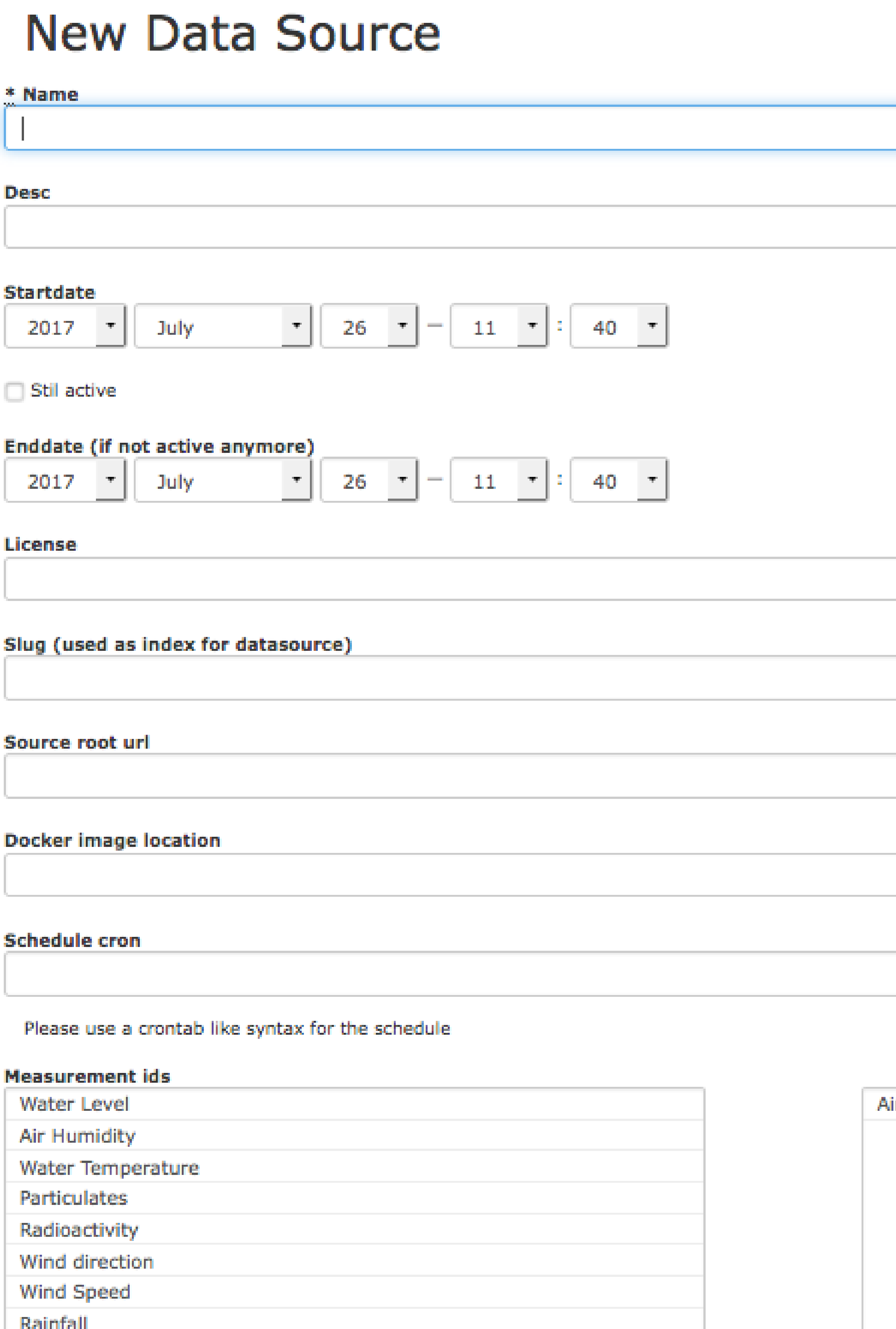
We chose not to store it in the same database as the actual sensor measurement data, but have a separate system instead. This provides isolation between our sources and the database, preventing changes in the choice of database or its schema from having any effect in how we store and maintain data sources.

The metainformation that has to be provided to our system prior to importing mainly consists of:

- Name of the source
- Start date from which the source provides data
- Whether the source is still active (i.e. data is actively being provided by the source).
- if not, the end date (last date for which data was provided).
- Under what license the data is published

Additionally we ask the user to provide additional information, which is useful for our system:

- If the data source is still active, at what schedule is new data published
- A *slug* that can be used as an id within our pipeline and in Elasticsearch
- The measurements provided by the data source (which actual measurements the data source collects)
- The URL location of the Docker container to be executed to import the data.



New Data Source

* Name

Desc

Startdate
2017 ▾ July ▾ 26 ▾ — 11 ▾ : 40 ▾

☐ Stil active

Enddate (if not active anymore)
2017 ▾ July ▾ 26 ▾ — 11 ▾ : 40 ▾

License

Slug (used as index for datasource)

Source root url

Docker image location

Schedule cron

Please use a crontab like syntax for the schedule

Measurement ids

Water Level
Air Humidity
Water Temperature
Particulates
Radioactivity
Wind direction
Wind Speed
Rainfall

Ai

Provide information to other system components

Validation Schema As described in the chapter about validation and insertion into Elasticsearch, we have a separate, distinct component that is responsible for validating the output from data importers.

To achieve this, a schema must be provided against which to validate. Since this schema itself is fixed, it could be hardcoded into the validator, without the need to make another HTTP request. We decided that this would be a bad idea for the following reasons:

– Sanity-checks: in addition to schema-only validation, the schema provided by this component can validate data-source-specific values (such as importer IDs, etc.), which could not be validated with a generic validator.
– Version management: schema changes over time would require changing the information hardcoded within the validator, and completely rebuilding and redeploying the validator component itself. Updating a record inside this component is far simpler than redeploying infrastructure, which is inherently more complex in an era where no downtime is expected.

Therefore we needed a place where said validation schema can reside. The web management system seemed to be the right pace for that, as it already carries metainformation about data sources and the data sources are registered there. So it is easy to provide the relevant schema information as well. The management system provides an API call to supply the validation schema to the validators which looks like this:

```
GET /data_sources/:id/getValidationSchema
```

The response looks like as follows (note that the const `source_id` would be replaced by the ID of the data source):

```
{
  "$schema": "http://json-schema.org/schema#",
  "title": "Data Source",
  "description": "A Data Source for Open Sensor Data from the CP project at TU Berlin. ",
  "type": "object",
  "properties": {
    "source_id": {"const": "source_slug"},
    "device": {"type": "string"},
    "timestamp": { "type": "string", "format": "date-time" },
    "timestamp_data": { "type": "string", "format": "date-time" },
    "location": {
      "type": "object",
      "properties": {
        "lat": {"type": "number",
                "exclusiveMaximum": true,
                "exclusiveMinimum": true,
                "maximum": 90,
                "minimum": -90
               },
        "lon": {"type": "number",
                "exclusiveMaximum": true,
                "exclusiveMinimum": true,
                "maximum": 180,
                "minimum": -180,
               }
      },
    "required": ["lat", "lon"]
    },
    "license": {"type": "string"},
    "sensors": {
      "type": "object",
      "items": [
      {
        "type": "object",
        "properties": {
          "sensor": {"type": "string"},
          "observation_type": {"type": "string"},
          "observation_value": {"type": "number"}
        }
      }]
    }
  },
  "required": ["source_id", "timestamp","sensors", "location", "license"]
}
```

Provide Information to the Elasticsearch API for query optimization As described in the data model section, our model is data-source-based and not measurand-based. For queries spanning across measurands, it will be necessary to have further information about the relationship between data sources and measurands.

As there were several possibilities as to where this information could be stored and managed, and how it would be provided, the easiest place to start with in our opinion was with the implementer of an importer, since he or she must provide information to us when registering the data source. The user therefore has to mark what measurements a data source contains upfront. (// TODO ref figure new data source)

In order to optimize querying, this information is provided to the API that wraps the search interface of Elasticsearch via an API itself. The information is stored in an indexed join table that holds the mapping between data source and the measurands it contains. As you can see in the class diagram (//TODO ref figure) of the relational system, queries in both directions are provided: getting all measurands for a data source, and getting all data sources that contain a given measurand. The corresponding routes look like this:

```
GET /data_sources/:id/measurements
```

Example:

```
GET /data_sources/blume_messnetz/measurands

[
    {"id":1,"name":"Air Temperature","desc":"","unit_category_id":"temperature"},
    {"id":3,"name":"Air Humidity","desc":"amount of water vapor present in the air","unit_ca
]
```

```
GET /measurements/:id/data_sources
```

An example request would look like following

```
GET /measurements/1/data_sources

[
    {"id":1,"slug":"blume_messnetz","license":""},
    {"id":5,"slug":"german_weather_service","license":""}
]
```

Provide configuration information to the deployment component Since the importer registration the only step in which the implementer has to provide information to our system, it should cover all configuration information needed to get an importer running.

7.2 Requirements

In contrast to nearly every other component we had to implement, the management platform did not have to meet as many criteria as a distributed cloud system in general. The data it contains is quite static and the number of requests that we expect is also quite low.

- Availability to other system components, that require the held information
- Quick response time (for query-optimization within the public search API)
- Ability to run asynchronous background tasks (for scheduling)

7.3 Architectural Details

The system was built using *Ruby on Rails* with *PostgreSQL* as a database. The reasons why we chose to use Rails are:

- Relatively fast development of an MVC web application

– Well established (over a decade), has therefore lots of resources — also for all kinds of extensions
– Extensive community support
– Very good integrated ORM adapters that could easily be exchanged for ODM adapters for document databases
– Offers the dynamic of Ruby as programming language

As mentioned before, the planning of the extent of the functionality offered by this system, was very vague. Besides managing metadata and providing an API for other components, further features were at least considered to be part of this system, like for example a scheduler that triggers the component that handles the deployment of importers. Therefore we chose to build this system on top of an infrastructure that can be easily extended in many directions and has nice-to-use database adapters instead of a more lightweight system.

While initially we also wanted to model a data source with its sensors grouped in sensor stations, this approach would bring immense overhead for configuring data sources in our system upfront, as a user would have to very exactly model a data source with all its sensors and sensor stations first. For big services like e.g. the German Weather Service this would be an enormous amount of work. Gathering this information would better be done by scraping the information present in Elasticsearch and translating them to geolocated information for all sensors/sensor-stations/datasources. In {//TODO ref figure} you can see the modeling for this approach greyed out.

The main metainformation about the data sources (besides metadata about the source itself) that we still needed within our system (the location of the sensor and the grouping of sensor stations is not essential for our system to work) and had to offer to the user registering a source were then:

– Information about the measurands offered by the data source
– Information about the main unit used for a measurand (see section Unit system // TODO ref section)

Future Improvements

Caching There is currently no caching solution implemented since the workloads during development phase were quite manageable. Also including a distributed caching system in our production pipeline seemed to be too high of an effort and would take up significant resources that would actually not be needed during development. We wanted, therefore, to use the limited and expensive resources we had for actual importing.

As the number of data importers grows in production, however, requests to the relational database would increase accordingly. Whereas scaling the database would allow us to avoid the bottleneck, adding a cache in front of the database to serve read requests would be sufficient to ensure performance without incurring the cost and added complexity of scaling. An important consideration, of course,

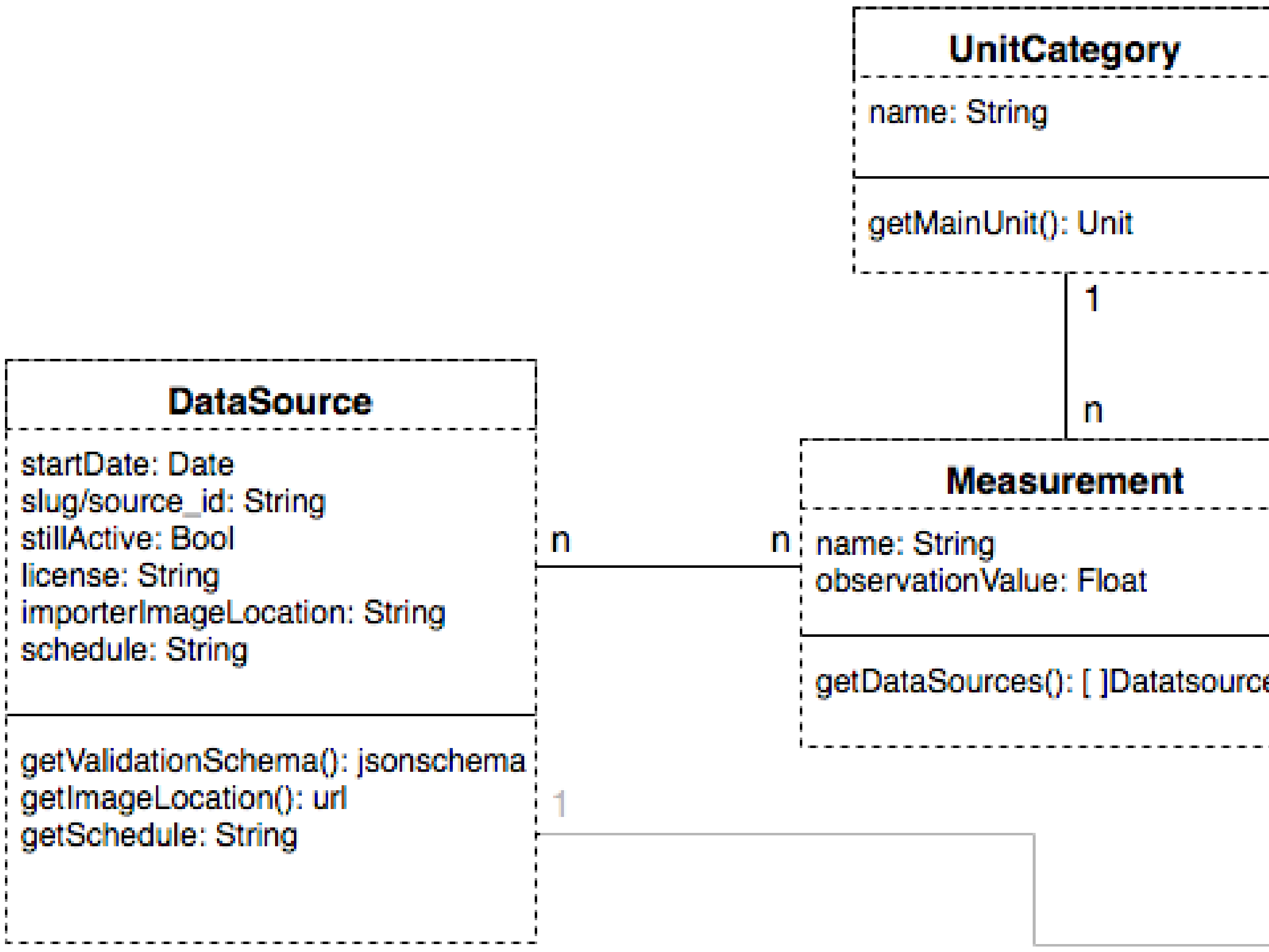


Fig. 3. UML Class-Diagram of the Relational Management System.

is the frequency with which data is updated (in our case low to none), thus making it a perfect candidate for caching. As a distributed caching system where read requests are very fast and possible on all nodes, Redis would be a good fit for this use case. Writing is quite expensive due to the replication method used by Redis, but because of the infrequent updates to our data, we are willing to accept the trade-off in exchange for very fast reads.

Exchange Scheduling information Due to not quite being able to reach every goal of our initial plan as to how the architecture should look like, the scheduler had to move to the deployment component (i.e. the deployment to the Kubernetes cluster). While this component should be responsible for deploying data importers, the information about the schedule should actually be provided to the web management system by the user. This is currently not happening. There would be several ways how to manage scheduling and/or deliver the scheduling information from this system to the component handling the scheduling.

- Having a background-processing component that acts as a scheduler within the web management platform. This would require an API to trigger the deploy on the component responsible for that, which we were not able to achieve.
- Having a microservice-like component that is only responsible for scheduling and triggering importers to be deployed. This would as well require an API on the deployment component.
- Leave the scheduling within the deployment component. This would also require an API, but just for receiving the general schedule, not for offering a hook to trigger an import.

The second option seems more granular and conforms better to our general microservice approach but would also require the most configuration and deployment effort. Including a scheduler in the web management platform would somehow violate this approach but still make sense, as this component could easily be integrated within Ruby on Rails and would be a standalone component within it.

7.4 Unit System

When it comes to gather and manage sensor data one topic that directly comes to mind is that of units. Uncountable units exist and while there are international systems like the *International System of Units (SI)* or the *metric system* it is hard to find one system, that fits all possible measurements in our case. Some of the reasons for that are:

- Not for all measurements there exists an standardized unit in unit systems (e.g. parts per million)
- While standardized systems have the advantage of offering *one* standard unit for a category, those do not have to be intuitive (e.g. using Kelvin for outside temperature probably will not be intuitive for many people)

Because of that we had to think of an own way, how units would be chosen, managed and how users would get information about them.

Further requirements Deciding on a strategy to model units, was a quite long process and not all requirements could be fullfilled. In this section we will discuss the most important requirements to later on explain for what approach we

1. Allow convertibility and ability to localize
2. It has to be understandable, in what unit a measurement is expressed and in which it was measured It is absolutely avoidable that measurements exist in the database, with a not understandable unit or even worse with a unit that differs from what the system supposes the measurement is expressed in. It is therefore advisable to enforce the user to handle units carefully
3. Each unit should only exist once. Typos, different expressions etc. shall not lead to confusion

Implementation With the requirements in mind we decided to organize units like so:

1. We introduced **Unit Categories**. Units themselves always belong to a unit category. The unit-category describes an entity for which measurements exist, which express their observations with one of the units of that category (See figure for Class Diagram // TODO ref figure).
2. Each unit has a **main unit** that we decide on. By calling the API or visiting the management platform a user can see, which the main unit is. Within our datastore we only use the main unit of a unit-category for expressing measurements.
3. Units are managed by admins receptively users with permit to do so.
4. We therefore have a curated list of the unit-categories and units
5. If there are units, measurements or even categories missing, each user can propose new ones. This proposals are also managed by the group of people managing the units.

Measurements are controlled on the platform itself to allow users to better propose new measurements, as this may happen more often. Units and the Unit categories however are managed in a *.yml* file. The syntax we used looks like following for one entry:

`From config/constants/units.yml:`

```
pascal:
  id: pressure_pascal
  name: "pascal"
  unit_symbol: "pa"
  unit_category_id: pressure
  notation: "1 <centerdot>
              <mfrac>
                <mrow>
                  kg
                </mrow>
                <mrow>
                  m
                  <msup>
                          <mi>s</mi>
                          <mn>2</mn>
                </mrow>
              </mfrac>"
```

From config/constants/unit_categories.yml:

```
pressure:
  id: pressure
  name: Pressure
```

The unit categories are pretty straight forward. For a unit there are some more possibilities. Besides declaring the unit symbol, the category it belongs to, its name you are allowed to use MathML to express what the meaning of a unit is. This is especially helpful with units that can be directly converted to each other. // TODO ref picture

The API of the web management system also provides calls to a) receive the main unit of a unit category and to b) get a list of all units there are for a unit category. Both of these information are of course aswell accessible from the frontend of the system.

```
GET /unit_categories/:id/getMainUnit
```

Example request:

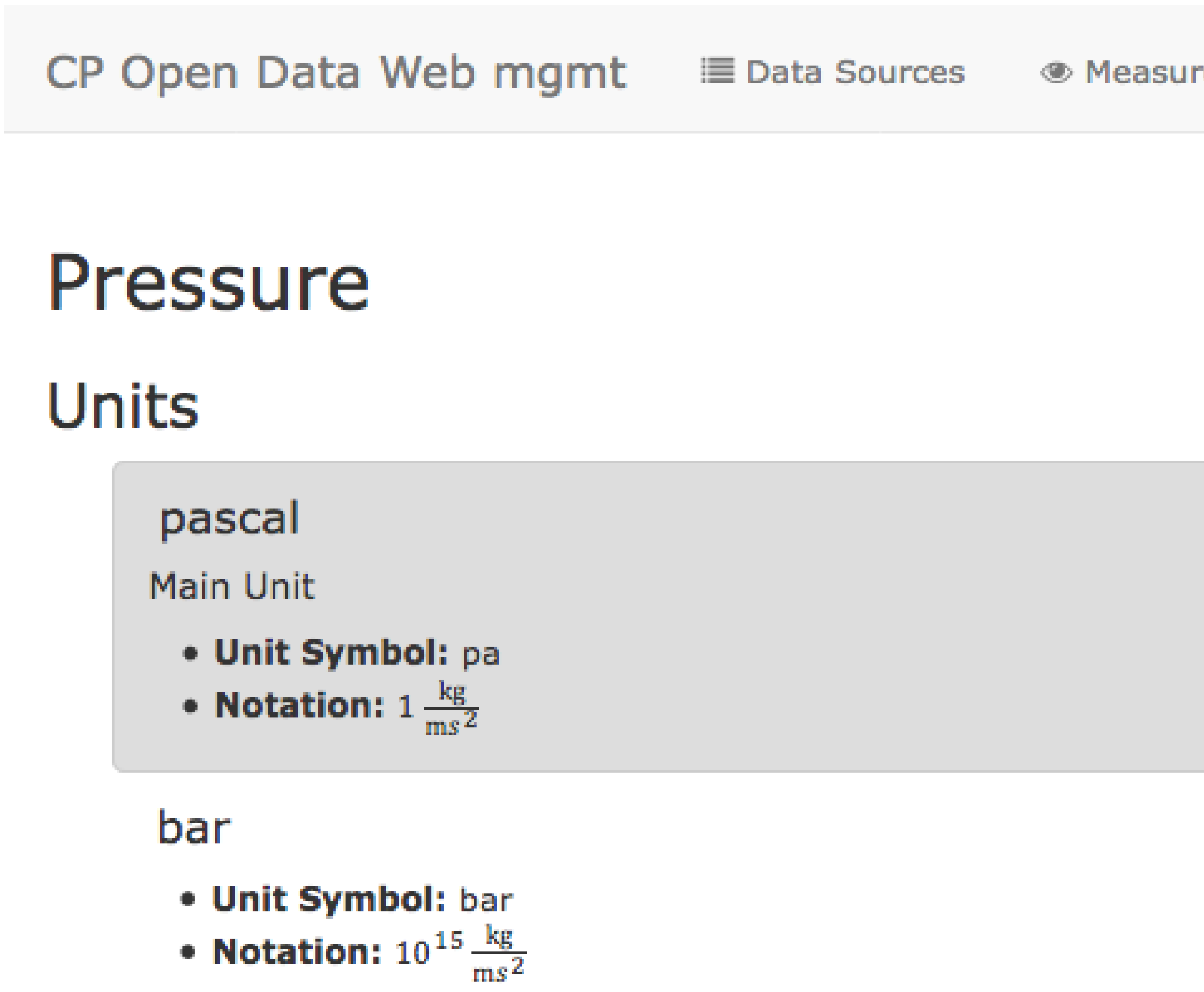
```
GET /unit_categories/temperature/getMainUnit
```

```
{
    "id":"temperature_celsius",
    "name":"celsius",
    "unit_category_id":"temperature",
    "unit_symbol":"C"
}
```

```
GET /unit_categories/:id/units
```

Example request:

```
GET /unit_categories/temperature/units
```



Pressure

Units

pascal

Main Unit

- **Unit Symbol:** pa
- **Notation:** $1 \frac{kg}{ms^2}$

bar

- **Unit Symbol:** bar
- **Notation:** $10^{15} \frac{kg}{ms^2}$

Fig. 4. Screenshot of the units of a unit category on the web management platform. You can see the main unit and have a notation on what the units express.

```
[
    {"attributes"
        {"id":"temperature_celsius","name":"celsius","unit_symbol":"C","unit_category_id":"
    {"attributes":
        {"id":"temperature_fahrenheit","name":"fahrenheit","unit_symbol":"F","unit_category_
    {"attributes":
        {"id":"temperature_kelvin","name":"kelvin","unit_symbol":"K","unit_category_id":"te
    [...]
]
```

Discussion This implementation has some advantages but also some disadvantages. In this section we want to take a closer look to both sides.

As we force the user to use our main unit, we can be sure, that all data in the database is of the same unit for a measurement type. Of course we cannot enforce that the user does convert measurements correctly or at all, but this would be considered as a faulty import, which in the end is the responsibility of the user. Of course an assessment of the correctness of data would be nice, but this is also hard to achieve and not within the scope of our project.

Our approach of having a curated list means some management overhead and possible longer implementation effort for the user if a unit conversion he or she needs is not yet available. Given the vast number of units in general and the lack of standardization in the way sensors report their data, giving a lot of latitude to the user to specify the units and the necessary conversions seems like the only reasonable way in which to approach the issue.

As the unit categories should be present after a short testing phase of a system, and a main unit exists with with, as the curators decide on one, the user should most of the time be able to register a source, when he wants to, as he only needs to know the main unit.

A big advantage of our approach is, that we kind of crowd-source the implementation of converters by this, as it happens during the ETL phase while importing (See //TODO ref chapter unit conversion) a source. This gives us a chance to achieve the following:

– Conversions can be reverted, as the converter used is stored with the data.
– Localization within our database can easily be done, as all measurements of a unit category have the same unit and converters are written the moment someone has to convert his source data to our preferred unit.
– By crowd-sourcing the implementation of converters they are also open sourced for reuse by other users. Having our own converters only in the system to convert measurements after they are in the database would not guarantee the reusability as importers and our database frontend depend on totally different things.

8 Conclusions

Authoriship

Version	Date	Modified by	Summary of changes
0.1	2017-07-19	Oliver, Andres	Working draft

8.1 Is Open Data really open?

– What sources did we "unlock"
– Data source types (ftp, http, REST)
– Different data formats (CSV, XLS, HTML)
– Different measurands (Temperature, air pollution, etc)

//`TODO` look into https://www.wired.com/1994/11/agre-if-2/ as a possible reference

8.2 Lessons learned

– Maslow's hammer: "when you have a hammer, everything looks like a nail"
– Database choice: Start with use cases & (types of) queries
– Test your ideas on real infrastructure rather conjecturing
– Balance using existing knowledge vs. investing time in learning new technologies
– Never underestimate overhead of cloud resource procurement, configuration & deployment
– Availability of "open" data is virtually inexistent for automated agents
– Most provide a GUI, but not the data

Team & Process

– Because English was not the native language of the majority of the team members, this led to difficulties with preciseness when having discussions and reaching consensus/conclusions
– Cultural differences (indirect speech, etc.) also contributed to protracted discussions and difficulties reaching agreement, in general.
– Team members tended to work separately. This in itself is not generally a problem, but in our case, due to lack of communication & synchronization with the rest of the team, led to end-products which sometimes did not work together well, or required significant efforts to integrate and harmonize. bugs or issues in the individual components were not discovered until quite late in the process (until deployment). Further, because the infrastructure deployment rested on the shoulders of a single member, this created a bottleneck and excessive load on this person.

– Development tools were established from the get-go, but somewhere around the midpoint of the project timeline most stopped using Trello to keep track of tasks and TODOs. Obviously this led to lack of visibility as to what was to be done, and contributed to the general displacement of team members into silos.

8.3 Future improvements

– What are the problems with the architecture as it is today – What was *not* implemented and WHY

– Postprocessing of indices/data into measurand-based hierarchy
– Optimize queries with metadata from relational database
– Ability to recover importer from failure via checkpointing
– User-defined time schedule for importers
– Auto-scaling of importers based on resource requirements

References