Project Documentation

Open Environmental Data
Cloud Prototyping Project

Written and submitted by:

Andres Ardila, Rohullah Ayobi, Oliver Bruski, Ahmad Jawid Jami, Amer Jazaerli, Nico Tasche, Paul Wille

Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany
firstname.lastname@campus.tu-berlin.de

Reviewed and published by:

Andres Ardila & Oliver Bruski

keywords — Open Data, Environemental Data, Big Data

Abstract

Despite a significant push in the past decade to liberate data, the world of static environmental ”open data” is giving way to the new wave of data available from IoT devices. However, despite the large amounts of potentially useful data that’s available, a platform on which enthusiast and organizations alike can donate and liberate their data, and where data scientists can take advantage of a way to query and layer all these data in new and unexpected ways is still an unrealized dream. With this project, we look at the current state of affairs of such static open data platforms, inquire about why they’re no longer serving their promise of openness and failing in their usability (and therefore usefulness), and introduce an early prototype for what such a system would look like if it were built with cutting edge cloud technologies.

Contents

List of Figures

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	Moved Organization content
0.3.0	2017-07-30	Oliver, Andres	Cleanup before hand-in

1.1 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/

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.

1.2 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 generate 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.

We therefore aim to create a tool that generates economic and social value through new and creative "layering" of data.

1.2.1 Building Blocks

The problem domain can be decomposed into the following architectural building blocks:

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

1.3 Requirements

The above objectives translate into the following requirements:

1.3.1 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.

1.3.2 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

1.3.3 Database

1. Ability to perform time series and geolocation range queries

1.3.4 Public API

1. RESTful
    - Stateless

2 Project Management & Organization

Authorship

Version	Date	Modified by	Summary of changes
0.1	2017-07-19	Andres, Oliver	Sections and initial outline
1.0	2017-07-27	Oliver	Outline & first written draft
2.0	2017-07-28	Oliver	Rephrased & extended team section
2.1	2017-07-29	Andres	Proofread

The main challenge of project management is to achieve specific results within a given scope, time, quality, and budget. At the end of every successful project stands a unique product, meeting the specified requirements. This chapter elaborates on the available resources, the development process and chosen approach. The resources that this chapter will focus on are the heterogeneous team and limited budget.

2.1 The Team

The project team consisted of seven masters students majoring in Computer Science, Computer Engineering, and Information Systems Management. Moreover, the team members are native of five different countries with different cultural backgrounds and native languages. Due to varying English skills and prior professional experience, some discussions resulted to be time-consuming. In addition, most team members work besides being students, hence schedules had to be aligned accordingly.

The team was organized into expert groups for the various building blocks, with one person in a coordinative role as the project manager. Thus, a hierarchy was formed within the group that was hard to enforce since, in the end, the team was formed of peers. In the beginning, two sub-teams had been formed: one that was responsible for the Importing Framework and data sources, and one for the Database and Infrastructure. As the project progressed, however, the roles of several members shifted. In the end, one member focused on the modeling and administration of data sources, another member focused on the infrastructure provisioning and deployment, the team responsible for the importing framework consisted of three persons, and lastly, one team two person team worked on the backend.

2.2 Constraints

Like every other project, this one was subject to several constraints. The very nature of a project makes it a temporary endeavor. Thus, the product had to be finished within eleven weeks starting April 27 and ending July 13. Additionally, towards the end of the first half June 1, an interim meeting was held. During these eleven weeks, three feedback sessions were held, one of them before the interim meeting and two after. After July 13, two more weeks were available to write documentation.

Moreover, monetary restrictions added an extra level of difficulty. While developing software consisting of several disparate building blocks, it is crucial to deploy and test the components individually and collectively. Due to a lack of an own infrastructure, the only feasible way to conduct testing —in functionality and performance— was to use a commercial cloud provider. Owing to no available budget, the decision for the US$100 student grant offered by Amazon Webservices (AWS) was made. Nevertheless, this added even more effort to receive access to the student grant and, furthermore, to also be able to use all of the services provided. Some members did not receive any free credit from AWS.

Moreover, the budget constraints forced the team to start and terminate the whole infrastructure every time a test was conducted. In other words, a significant amount of time was necessary to create the test environment each time, which put additional strain with regards to the already strict time limit. Hence, a significant effort was put into automating processes in order to provision the infrastructure and deployment of components, and then shutting them down to prevent the budget from depleting.

Having dedicated infrastructure from the beginning would have allowed to focus more on the actual challenges of the project itself.

2.3 Development Process & Approach

To best approach the given challenges and requirements, we decided for an agile development process. To be more specific, we used a stripped-down Scrum approach. In this approach, the distinction between different roles was not strict so that every team member held simultaneously the role of a product owner and developer for his specific building block of the global architecture. The same is applicable for the role of Scrum Master, which was shared among the whole team, but still was mainly the responsibility of the Project Manager. Specifically, this means that every team member was encouraged to ensure an agile process and manage issues. Furthermore, the workflow was adapted to our specific scheduling needs so that we convened a sprint meeting weekly with an additional weekly interim meeting and omitted the daily scrum.

To keep track of every issue and the global progress, we used a Trello board. Trello follows the Kanban model and arranges issues into lists. We created four lists to mirror the lifecycle of issues. The first list —product backlog— is where requirements, issues, and general tasks that the product must satisfy were created; this list was populated continuously as the project progressed. The next lists included: tasks due until the next sprint meeting —the sprint backlog—, tasks that were currently being worked on, and finally, finished tasks were archived.

Slack was the chosen medium of communication, because it allows to distinguish between independent topics by using channels. Additionally, a shared calendar was used to remember deadlines and a shared file system to store our meeting minutes. More importantly, we used GitHub to store the code written. We formed an *organization* and created a separate repository for every component.

In order to integrate the tools we used as well as to automate workflows, a service called Zapier was used. We used it to receive notifications about changes on GitHub and Google Drive, and the progress made on Trello. Though, after the trial ended the functionality was limited so that only notifications about GitHub updates were sent.

3 Survey of Current Landscape

Authorship: Written by Andres Ardila

3.1 OpenSensors.io

https://www.opensensors.io

3.1.1 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.

3.1.2 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. Air quality and traffic data available "free".

3.1.3 Customers

Partners are hardware manufacturers and design firms (OEMs) looking to deliver client projects.

3.1.4 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."

3.2 Plenario

http://plenar.io/

3.2.1 Features

> "Plenario is a centralized hub for open datasets from around the world, ready to search and download."

- Open platform on which to add open datasets in CSV or ESRI shapefiles
- Only requires data to have time and location information in the datapoints
- Modern open source data visualization for timeseries geolocated data (heatmap, line charts)

3.2.2 Customers

Cities who already generate and publish open data and wish to provide a rich interface to users to visualize data from various datasets.

3.3 Analysis

Plenario is precisely the next step forward from static data platforms like CKAN. By requiring that data must contain time and location information, an elegant, unified, and intuitive visualization is immediately made possible. Unfortunately it offers no rich querying model, other than filtering by date ranges, but the concept of 'layering' is already present, by allowing the user to visualize data from more than one data source for analysis on the user interface. This is definitely a step in the right direction and most importantly, it's open source.

3.4 PubNub

https://www.pubnub.com/

3.4.1 Features

> "The Programmable Data Stream Network Low-latency messaging. Massive scale. Ready for the real world."

As an IoT end-to-end publish-subscribe messaging and streaming platform, we are concerned only with their 'Storage & Playback' product.

- Scalability: 15 globally replicated points of presence transacting over 1.5 trillion messages for 300 million unique devices per month.
- Unlimited storage
- Configurable retention period
- Message-level granularity

3.4.2 Analysis

PubNub provides robust messaging infrastructure, which coupled with their storage solution could be seen as providing (at least at a very low level) the piping for the import part of our system. The programmer is given an lambda-like programming model in which each event can be processed. For live streaming sensor data this could be a viable solution, as the lambda could be seen as performing the data integration task of schema conversion between the sensor data model and our internal schema to be then be stored. For historical importers, however, this would not be an ideal model. Unfortunately, since it is not open source, there are no details available as to the storage engine or internal architecture, which could be of value for our development as inspiration.

3.5 Talend

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

3.5.1 Features

Talend is primarily a data integration solution. Their offering for sensor data revolves around providing a “an easy-to-use graphical environment, [where] developers can visually map sensor data sources and targets, and quickly perform complex transformations and analyses to produce actionable intelligence and drive performance improvement.”

Talend also provides pre-built connectors for Hadoop database solutions like Cassandra, CouchDB, Couchbase, HBase, MongoDB, Neo4J and Riak .

3.5.2 Analysis

Talend’s offering concerns the data integration aspect of our system, so it is not comparable when considering the system as a whole. Talend is positioned in the Spark + Hadoop ‘Big Data‘ space and as such targets the leveraging of data to generate analytics. While their data connectors are an attractive feature in general, our users most likely won’t be providing data through a database connection, but rather through files, so it’s unfortunately not a particularly good fit. Their data integration IDE would certainly be a nice feature for those users who are not necessarily programmers but rather simply enthusiasts; a graphical interface to perform a simple schema transformation would be sufficient for them. Further, storage is provided as a subscription product as opposed to being an open source solution which we could peer into for inspiration.

4 Architecture

Authorship:
Written by Oliver Bruski
Proofread & edited by Andres Ardila

The challenge of software architecture is to create a structured solution that meets all given requirements and constraints for a domain problem. It presents an abstract structure on the system and includes the building blocks of the solution with their interaction without focusing on the low-level implementation details like interface design [6]. Moreover, the system architecture presents a blueprint not only for the system but for the project as well. The architecture can be utilized to divide the development team into groups. The process of producing a system architecture starts with a high-level view of the whole project and is incrementally broken down into smaller subsystems [4].

In order to design a reasonable architecture, it is necessary to know the domain of the problem. Think of the domain as the problem area and the domain model driven design as the solution for this specific problem. The domain model represents an organized and structured view on the problem. Consequently, the domain should be defined in the beginning of a project to avoid misunderstandings and put everyone on the same page. The domain-driven design focuses on understanding and interpreting the specific aspects of the given problem.

This chapter will elaborate on the motivation and the decisions we made. Further, we discuss existing architecture for similar problem statements. The main part will of this chapter will present the created architecture and how it evolved over time. Finally, we conclude the chapter with a critical analysis of the architecture and discuss possible future improvements.

4.1 Motivation

Due to the distributed nature of the project, the target was to build upon loosely coupled components. For this purpose, a microservice architectural design was the best fit. Using a microservice approach allowed to decouple the system and divide it into independent modules. Microservices interact through specified interfaces in contrast to a monolithic system, where the different modules depend on each other due to inheritance of classes, for example. By communicating just through specified interfaces, no unintended dependency will be formed. Every interaction between services has to be implemented explicitly [9].

A positive side effect of the loose coupling of components is that it allowed to divide the team according to the main building blocks. Each component was the responsibility of a different group. Furthermore, the absence of strict dependence and inheritance among services allowed to use several technologies and programming languages. Thus, every group was able to use the language that suited its purpose best. In addition to independent development, we were able to conduct tests independently from other components. This was necessary due to the limited budget for infrastructure and deployment. Hence, we were able to test every component stand-alone before testing the whole system.

Another benefit of a microservice approach is that nearly every component is independently scalable and load balancing can be accomplished using smart routing. Furthermore, by creating stateless components, horizontal scalability was easily achievable. By replacing point-to-point communication with a distributed messaging system between the different building blocks, we were able to improve scalability even more and decouple even further. Firstly, the queue introduces a separation layer between the components, which allows to easily exchange one component without affecting the others. Furthermore, the queue can work as a buffer to protect the components from temporary peaks in workloads.

Obviously, there are some challenges that have to be resolved to harness all the benefits of microservices architecture. In contrast to a monolithic solution, every microservice can be deployed independently. Also, this is beneficial for our limited budget and allows to test and deploy the components separately, but it comes at a price. A complex system can consist of many microservices that have to be deployed individually, thus it is crucial to automate the deployment process. If implemented correctly, a continuous delivery pipeline presents huge benefits, not just for microservices. We adapted the continuous delivery pipeline introduced by Wolff and customized it to fit our workflow. (Commit – local test – system test – performance test) [9]. It started with a commit to GitHub and was followed by local tests. Here again the microservices approach presented its benefits, since it allowed for independent testing. Unfortunately, only towards the end of the project were we able to conduct system tests and examine the functionality in an end-to-end workflow. Nevertheless, through meticulous local testing and loose coupling of the components we were still able to prevent huge changes to components when other components changed. Finally, we conducted performance tests for the complete system to ensure high performance and prove scalability. Lastly, the performance of single components was also evaluated to avoid bottlenecks.

In addition to the challenges presented by the deployment process, operating a microservice system presents its own set of challenges. Every component is monitored independently and the logs have to be gathered in a central storage. Logging is necessary to ensure availability of each service. Should a service terminate unexpectedly, the administrator has to restart the service to ensure the performance of the whole system. The system should nonetheless be still available during the downtime of a single service with some possible losses on functionality. Chapter 5 will discuss how we enabled recovery and fault-tolerance of the system.

The next section will presented the specific requirements for the given problem statement and how microservices architecture presents a solution for it.

4.2 Requirements for the System

To create a system that is able to resolve all the presented objectives, we identified the following architectural requirements:

- **Scalability**
- **Extensibility**
- **High Performance**
- **Fault-tolerance**

Systems are expected to grow over time and handle increasing loads. Thus, we need a system architecture that allows to scale effortlessly and benefit from economies of scale. Besides, by following a microservices architecture components should include as little functionality as necessary and in addition, not share any state to avoid synchronization penalties. Finally, single-points-of-failure must be avoided to ensure availability even if a single component fails [5].

Next, hoping that the system under development will provide value to the open data community and more and more people will use it, it is necessary to handle increasing

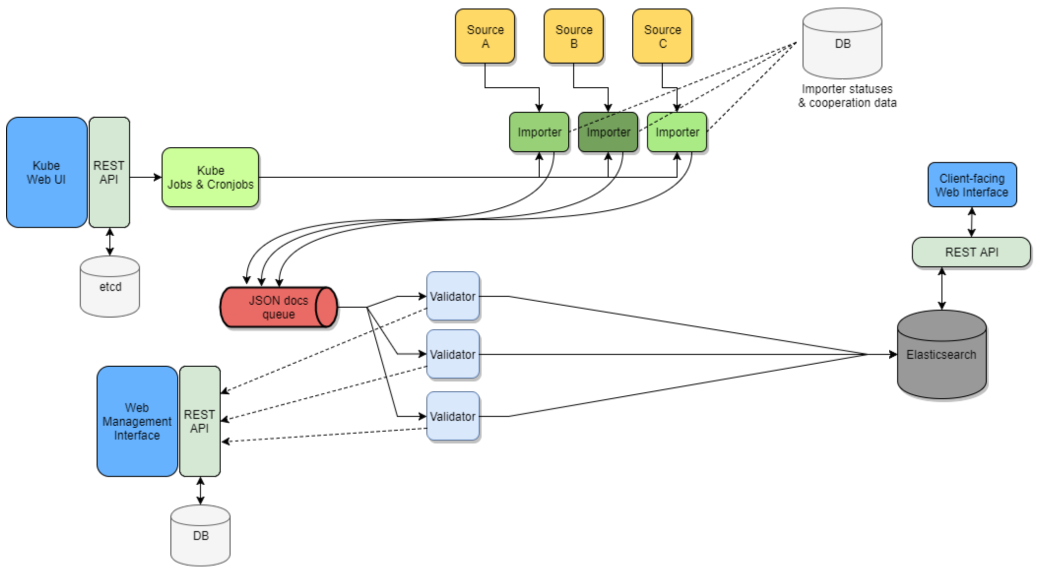


Figure 1: Final Architecture

workloads and ensure high performance even under peak loads. Additionally, the architecture must be fault-tolerant and ensure good performance even if some nodes fail, obviously losing some functionality.

Moreover, the system is expected to be extensible to allow for contributions by the community. Specifically, we offer an importing framework in order to simplify sharing your own environmental sensor data on our platform. Thus, the architecture must be designed in a way that allows to incorporate modules from other developers without significant changes to the system.

Finally, the goal of the system is to provide an end-to-end solution for open data enthusiasts to share their collected data as well as provide a platform from which data can be retrieved and combined with data from other sources to add value to the data through smart combinations. This requirement decomposes into four steps:

1. Collect and ingest data from various heterogeneous sources
2. Process and transform the data
3. Provide persistend storage for the data
4. Offer a uniform interface to retrieve all data contributed

4.3 Design Decisions

This section will discuss the decisions concerning the architectural design on a higher level. Decisions about specific technologies and components will be presented in the following chapters. To tackle all of the mentioned requirements, we decided on the architecture depicted in Figure 1.

The final architecture is composed of five building blocks:

- Importers
- Messaging System
- Consumers

- Database
- Infrastructure Monitoring & Scheduling
- Data Source Metadata Management

The main goal was to create components that are stateless and which do not need to synchronize. This allowed our system to scale horizontally without additional effort. To avoid bottlenecks, we chose components that are distributed by design. Moreover, as discussed earlier, every building block should be independent from each other. Thus, every building block could have been developed independently and be easily exchangeable if and when necessary. The components communicate through a technology-agnostic interface or are separated by a distributed messaging system. Now we will discuss how we designed our building blocks to satisfy the presented requirements.

4.3.1 Importers

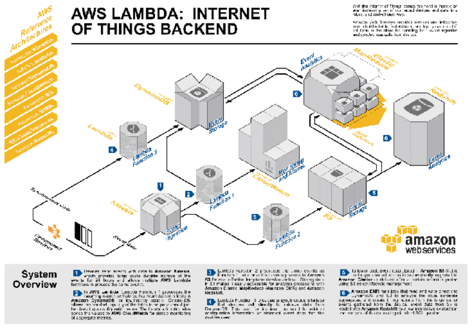
The system is supposed to handle all kinds of heterogeneous sources. At this point we focus on the heterogeneity of the data size and frequency rather than the different protocols and interfaces. Since most sources have a unique frequency at which they update their data and the amount of measurements differs between sources, the system should allow to schedule the importing task accordingly. Hence, every importer runs as an independent service with a specific lifecycle (run-time and frequency of execution) and specific scope (timeframe). Upon finishing its job, the task terminates and the resources are released to allow for maximal resource utilization. Following a microservice design, every importer manages its state independently from the system and can restart from a checkpoint in case of failure. Using such a design allows for decoupled services that can be spawned independently, are fault-tolerant and easily scalable by adding a new importer for a different time period or source.

4.3.2 Messaging System

To decouple the components even further, we introduced a queuing system to transport and buffer messages between the importers and consumers. Since this building block is crucial to ensure scalability and should protect the system from workload peaks. Following the “think big” approach, we imagined a system capable of handling millions of concurrent requests. In order to achieve such a high throughput, the system has to be designed in a distributed manner. To additionally introduce fault-tolerance, we need a service that replicates data automatically to compensate for crashed nodes. Automatic partitioning would be a further benefit to assist load-balancing. In contrast to a monolithic application, by decoupling the components with a distributed messaging system, the autonomous services can be deployed independently and enable fault-tolerance for the system as a whole.

4.3.3 Database

When it comes to the choice of a database in addition to the requirements that applied for the other building blocks the schema of the data is what is of major concern. The data model and the specific choice of a database technology will be discussed in more detail in Chapter 9. At this point the decision for relational or NoSQL databases has to be made. The designed system is supposed to interact with many heterogeneous source and data models. Thus, making it hard to fit into a relational model. Additionally, the emphasis of our system is availability rather than consistency, which is why we can


AWS Reference Architectures
AWS LAMBDA: INTERNET OF THINGS BACKEND
System Overview
amazon webservices
(a) AWS IoT Reference

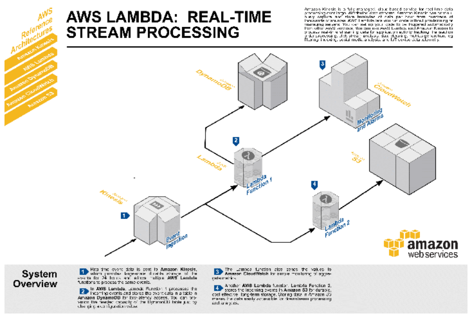

AWS Reference Architectures
AWS LAMBDA: REAL-TIME STREAM PROCESSING
System Overview
amazon webservices
(b) AWS Stream Processing Reference

Figure 2: Researched AWS Reference Architecture

spare transactions in order to achieve higher availability and performance. Most NoSQL databases are designed to partition and shard the data to increase performance and sacrifice consistency [5]. Moreover, asynchronous replication is good enough in terms of consistency, when the availability of the system is increased.

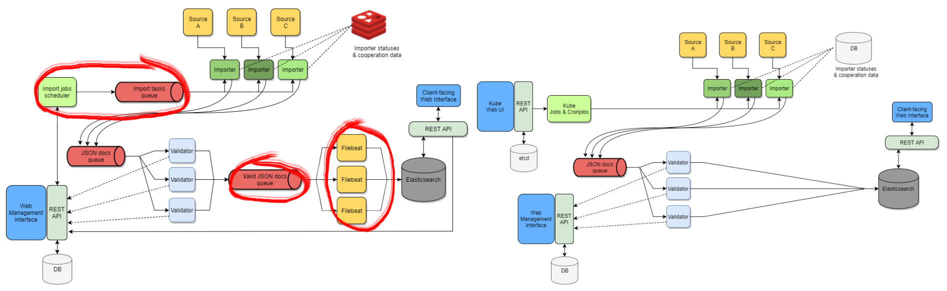
Concluding, the choice in favor of a NoSQL solution was the right decision. NoSQL databases allow for higher flexibility and are designed to scale across several nodes.

4.4 Existing solutions

It is always advisable to research existing solutions and contributions to the given problem. By doing so, first, one avoids doing the same work again and emphasize that the proposed work provides added value. Furthermore, one can learn from existing solutions and approaches to solve the given problem. In this section we will discuss the existing architectures to resolve the challenge of collecting huge amounts of data and process and store it persistently.

Our problem statement is to collect open sensor data, process it in batches, store it persistently and provide it to the user through a unified interface. The Amazon Web Services (AWS) Lab offers an extensive collection of reference architectures for their cloud platform. Even though it is built on top of the AWS cloud service, most of the offerings by AWS can be replaced by open source solutions with similar functionality in order to run on different infrastructure. We identified three proposed reference architectures that could fit our problem statement. Figure 2 shows the IoT Reference Architecture and the Stream Processing Reference Architecture on AWS. Even though the shown reference architectures do not mirror our whole pipeline, they do overlap in some aspects, which makes it interesting to investigate them further. The third reference architecture, Batch Processing with ECS suggests more of an approach to employ containers with AWS' container services. Since we wanted to create a provider independent platform, we did not pursue it any further. Though we did adapt a container approach using Docker to abstract away from the underlying infrastructure.

Our use case is comparable to a IoT service. The main difference is that we do not ingest data from a stream into our system but rather in batches at scheduled times. Nevertheless, the data arrives in a myriad of different formats and in addition, depending on the success and popularity of the system, at some point in time the batches will be so frequent that it might resemble a stream. This workflow is similar to the one we wanted



(a) Initial Architecture (b) Final Architecture

Figure 3: Evolution of Architecture

to build: The data enters the system, gets processed by a small and simple component and then is stored for querying.

The stream processing architecture is a little simpler. Though here the pipeline is comparable to the workflow that we employ. Besides the low-latency storage in DynamoDB, an archival storage in S3 is created. We believe that retrieving historical data together with new recent data is one aspect that adds value to the retrieved data, thus we serve the data independently of the arrival time in our main storage solution. Nevertheless, it presents a possible solution on how to store data that is not frequently queried when the stored amount is hard to be processed by a single engine.

All the reference architectures mentioned (plus more) can be found at `https://github.com/awslabs`. The following list includes the links to the discussed architectures:

- `https://github.com/awslabs/lambda-refarch-iotbackend`
- `https://github.com/awslabs/lambda-refarch-streamprocessing`
- `https://github.com/awslabs/ecs-refarch-batch-processing`

4.5 Evolution of the architecture

As the project evolved and the understanding of the domain increased, the architecture evolved as well. It is the nature of an agile project to evolve over time when the product owners add features to the system. This section will present an overview of the evolution of the architecture and the changes over time.

The changes in the architecture are depicted in Figure 3.

Basically, the architecture was updated in two places. First, the scheduling logic was exchanged. In the final architecture, we rely on Kubernetes and cron jobs to schedule the importing tasks. We decided for Kubernetes to manage the underlying infrastructure, so it was feasible to use it for scheduling as well. The underlying infrastructure will be discussed in more detail in Chapter 5, but since the placement and deployment of components is done with Kubernetes, we decided to use it to schedule the importers.

Additionally, *filebeat* and one messaging queue got removed. To reduce the latency of the system, the validator and filebeat were merged into a single component: the consumer. This changed made the messaging queue in between those components become obsolete. At this, point we coupled functionality and added dependency but the increased performance made up for it.

Summarizing, we exchanged one building block with a better alternative and shortened our pipeline in order to increase throughput.

4.6 Discussion

We designed a fault-tolerant, highly scalable and available architecture based on microservices design. The architecture is composed of several independent and loosely coupled services that interact through technology-agnostic interfaces. By doing so, we could use different technologies for every component and develop or replace each building block independently.

On the flip side, we created a highly complex architecture consisting of several technologies and services, which makes it hard to be understood by a single person. Furthermore, the deployment and operation of the system has to be automated otherwise an administrator might reach his or her limit quickly. We offer scripts to deploy the system onto AWS, but this can be applied to every other cloud provider or a private cloud with minor changes.

4.7 Future

No system is perfect, hence, in this section we propose some possible future updates that might improve the proposed solution. The major value of the designed system is the possibility to retrieve open environmental data through a uniform interface. In other words, the query performance of our database solution is the critical concern. One possibility to increase query performance for complex requests might be the implementation of an interaction between the public API, exposed to the user, and the Data Source Metadata Management System. By doing so, the API could learn which indexes include the data to answer a specific types of queries in contrast to answering the query in a naive way without any optimizations.

A different approach to improve the system might be the introduction of a serverless component. Though this proposal only applies to a solution deployed on public or hybrid clouds that offers such services. Such a serverless service might replace a functionality or module of an importing task. For example, one could use it to for unit conversion. On the other hand, it might be necessary to perform benchmarks and evaluate the added value through a serverless component. Since we release resources immediately after a job finishes, it is hard to tell if it really is more cost efficient without proper testing.

5 Infrastructure

Authorship:
Written by Andres
Component developed by Amer

Infrastructure undoubtedly plays a critical role in any cloud-based system. In this section, we will discuss the requirements which were specific to the choice of cloud platform and orchestration as well as all activities surrounding deployment to the cloud.

5.1 Requirements

Based on the particular factors discussed, we identified the following points as being important requirements as part of our infrastructure:

1. **Platform agnostic components**: Our infrastructure must be deployable to both public and private clouds. Therefore, making use of provider-specific features was not possible.
2. **Budget**: The choice of cloud provider should allow us to prototype and test in a shoestring budget (US$100 for the whole semester).
3. **Scalability**: The ability to have scalable infrastructure is a critical requirement. This applies to all architectural building blocks. Our infrastructure, therefore, must support scalability out-of-the-box.

In addition, the general requirements for the system with regards to scalability, performance, etc. also apply to this component, as mentioned.

5.2 Evaluation Criteria & Decision-making Process

5.2.1 Cloud Platform

The cloud-agnostic nature of our system would have allowed us to deploy prototypes to any cloud provider to which we had access. Due to the unavailability of cloud resources to our project, however, we had a single choice when it came to deciding on the platform on which to deploy our infrastructure, namely AWS since it was the only platform for which we had credits. Conducting a thorough comparison of cloud platforms which we subsequently wouldn't have been able to use, did not seem like a good use of our time.

5.2.2 Orchestration

Originally an internal Google project, Kubernetes was opensourced via a donation to the Cloud Native Computing Foundation (CNCF). Among others, its core functionality includes support for deployment, maintenance, and scalability of applications.

5.3 Implementation Details

5.3.1 Orchestration

Given its prominence in the cloud arena and the fact that it is open source software, we used Kubernetes for orchestration of cloud components. Having containerized data

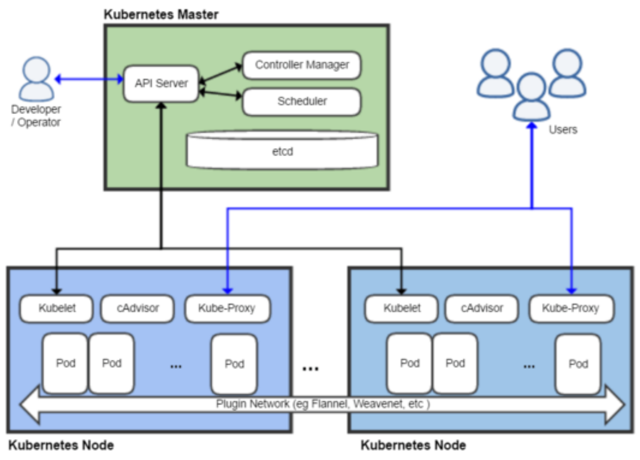


Figure 4: Kubernetes architecture diagram [7]

importers was a requirement from early on in order to accomplish scalability, and Kubernetes supports the creation and monitoring of containers natively via its *Pod* abstraction. Support for Docker containers out-of-the-box was important feature for us, as it encapsulated the run-time needs of our data importers. Since they needed to be scalable, we envisioned from the beginning that these data importers should be wrapped in some abstraction which could be launched by some controller or orchestrator in our cloud. Docker also gives us the freedom to develop in any language or technology, so far as the container is well documented it can be started and called when it is needed. Kuberenetes *Replication Controllers* then define how many pods or containers need to be running.

In addition, the building blocks of our architecture such as the queue and the database, which must be guaranteed to be up and running, were declared as Kubernetes *Services*, which provide additional self-healing, should these critical components crash.

Additionally, internal DNS provides ease-of-use by being able to refer to components by a friendly name as opposed to IP addresses. As mentioned in the Architecture section, scheduling was realized via simple cron job (see Figure 5).

Finally, in addition to the command-line interface provided by kops, a state-of-the-art web interface is provided out-of-the-box, making for a superior user experience.

5.3.2 Configuration

One of the main advantages of Kubernetes is the ability to specify behavior of common abstractions via configuration alterable at run-time rather than by having to program components responsible for carrying out hard-coded logic. Configuration is provided by YAML files for registering crons, data importer job runtime configuration, and service-specific configuration necessary for inter-node communication.

5.3.3 Deployment Automation

Due to the very specific constrains that our project found itself in, a disproportionately large portion of the effort went into managing infrastructure. Having no budget meant

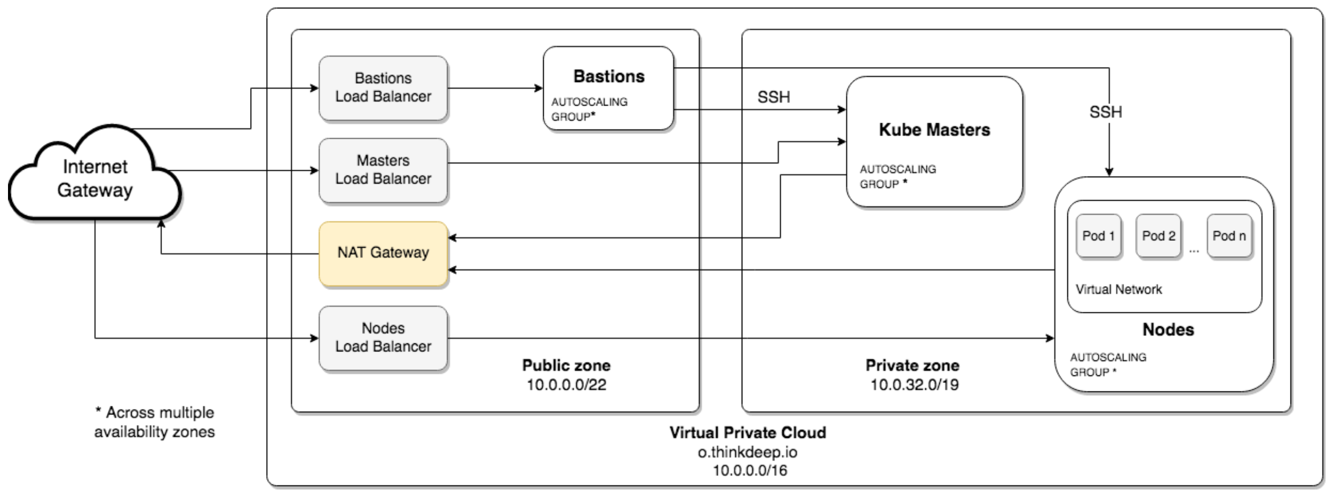


Figure 5: Kubernetes Topology

that whatever infrastructure was deployed had to be immediately torn down after it was no longer needed.

Difficulties notwithstanding, this meant that the automation process for deployemnt got well refined, with robust scripts, and the addition of Kubernetes Operations (kops) and its native support for AWS translated into an even smoother deployment experience.

Spot instances also played an important role in keeping to the budget, and support for these were was integrated into the scripts, which allowed us to monitor and specify the price for spot instances.

6 Extensible Data Import Framework

Authorship

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

6.1 Overview

The Open Data Platform which we have built for extracting, transforming and loading open sensor data is made up several 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.

6.2 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.

6.3 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

6.3.1 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

6.4 Design Decisions

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

6.4.1 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

6.4.2 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

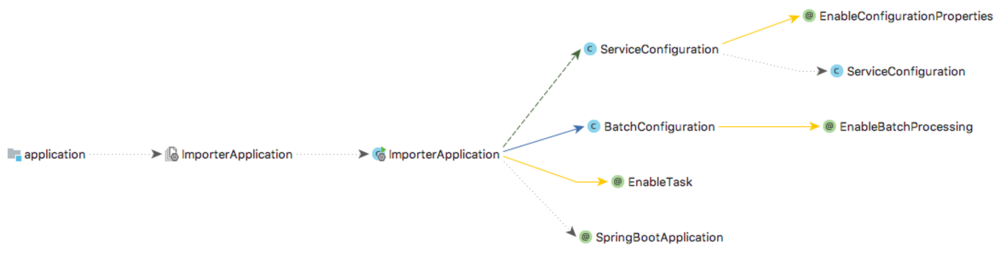


Figure 6: Importer Application Overview

- 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.

6.5 Framework Structure

Figure 6 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).
- **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 re-usable classes across all importers are defined. The following other components are included in the 'library' module.

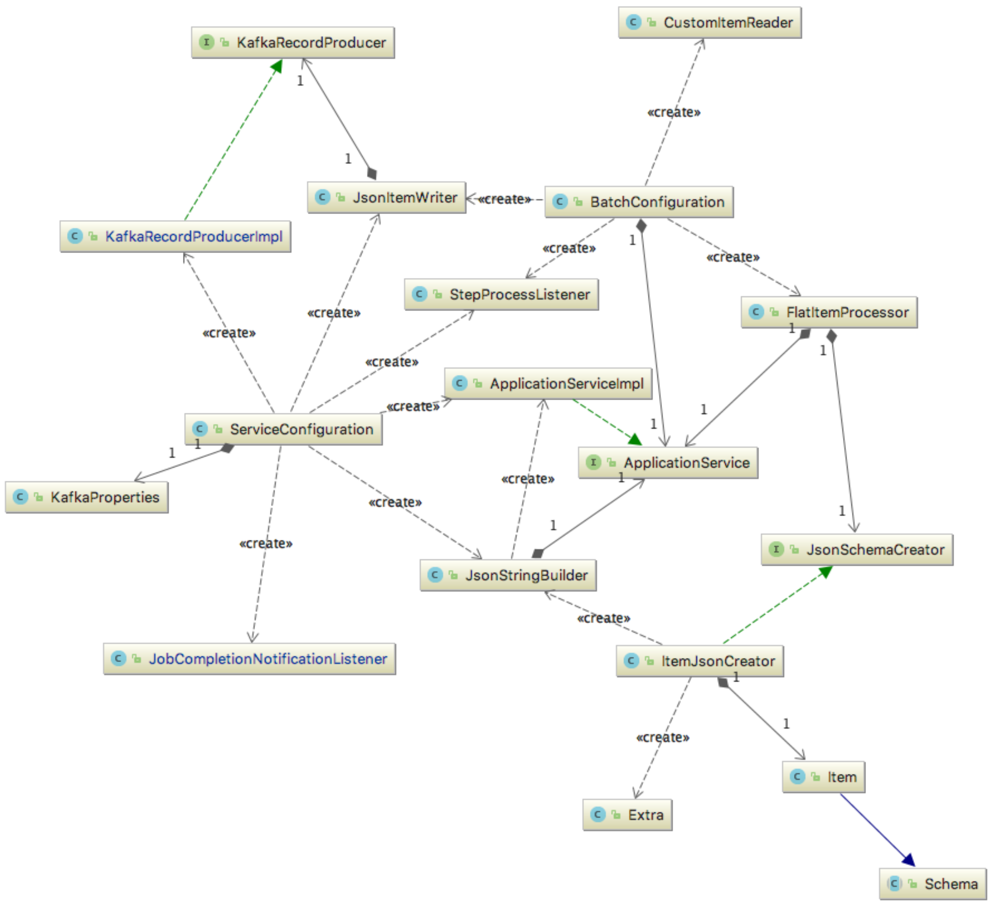


Figure 7: Relation of classes of a Simple Flat File Importer

- **ApplicationService:** Includes some generally used methods to bring facility for importing.
- **JsonItemWriter:** Asks `KafkaRecordProducer` to write individual JSON objects to Kafka queue.
- **KafkaRecordProducer:** Is the class where the items are written to Kafka queue.
- **Schema:** Each importer should have at least one model class which has all the necessary fields for reading records from a source; all these classes are extended from Schema.
- **JsonStringBuilder:** The class which contains functions for creating a json string according to our defined schema.

Figure 7 shows a more detailed view of classes inside a simple flat file importer and how they are related together.

6.6 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

6.7 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

6.8 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, HTTP and FTP data source types, Delimiter-separated value (DSV): CSV, TSV; HTML; XLS; XLXS, etc.

6.9 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.

6.10 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.

7 Units of Measurement

Authorship: Written by Andres Ardila

7.1 Overview

Since the same physical quantity can be represented in different units of measurement — both meters and miles represent lengths, for example— storing measurements of physical quantities inherently introduces the complexity of these different representations. While the topic is certainly quite interesting, especially when considering how sensors process and convert signals to digital values for a measurement, we shall limit the scope of our discussion here 1) to how our system deals with the different ways in which the same physical quantity can be represented, and 2) to the facilities provided to users by the system to convert their measurements from one unit to another.

7.2 Requirements

From early on in the analysis phase, it became apparent that in order to deal with the heterogeneity of incoming data and to fulfill the requirement of providing a uniform query interface for the data, a global schema would have to be introduced and enforced all imported data. That is, someone who is performing a query would expect the data to be structurally and semantically consistent, even if it came from different sources. Say our platform offered ambient temperature readings; this data may come from different sensors manufactured by different companies and which provide data in different formats. In order for the data to be usable for analysis, I, as a data user, would not expect to have to convert readings from the U.S. to Celsius in order to be able to compare it with data from Europe, or from degrees Kelvin if a particular sensor manufacturer has calibrated its devices to report ambient temperature in this unit.

Having standard units for a given physical quantity means, however, that data importers must take care to convert all units to the standard for the platform. In order to reduce friction for our data donators, we also wish to provide facilities to convert their units to those required by our system. With this in mind, we formalized our requirements with respect to units as follows:

- all measurement values stored and reported by our system must be accompanied by a unit of measurement
- any measurement values which underwent unit conversion must include information about the converter so that it may be reverted, if necessary
- provide extensible facilities to users to convert units within their importer packages to the required unit defined by the system for that physical quantity

7.3 Survey of Previous Work

7.3.1 Standards

ISO ISO 1000 – *"SI units and recommendations for the use of their multiples and of certain other units"* was first introduced in 1981 and revised in 1992, but withdrawn and superseded by ISO(/DIN) 80000 in 2009. The standard is under the ISO/TC 12 technical committee, which is responsible for

> "Standardization of units and symbols for quantities and units (and mathematical symbols) used within the different fields of science and technology, giving, where necessary, definitions of these quantities and units. Standard conversion factors between the various units."

In addition, now-withdrawn ISO 2955:1983 *"Information processing – Representation of SI and other units in systems with limited character sets"* deals with encoding unit symbols for machine processing.

Unfortunately, ISO standards are not available free of charge, so their relevance and usefulness to our project could not be evaluated.

ANSI X3.50 The 1986 standard *"Representations for U.S. Customary, SI, and Other Units to Be Used in Systems with Limited Character Sets"* deals with the symbolic representation of the units, and as a result is not of particular interest to our objectives, namely:

> "This standard was not designed for [...] usage by humans as input to, or output from, data systems. [...] They should never be printed out for publication or for other forms of public information transfer."

NIST 811 The *Guide for the Use of the International System of Units (SI)* from 1995 and updated in 2008 provides a comprehensive reference regarding SI and units in general aimed to assist scientific paper authors from the National Institute of Standards and Technology (NIST). The guide proved useful in condensing the wealth of information that is SI and the variety of units and formats in an straight forward, complete yet concise document.

Unified Code for Units of Measure (UCUM) Based on ISO 80000, UCUM's "purpose is to facilitate unambiguous electronic communication of quantities together with their units." Like ISO 2955 and ANSI X3.50, its focus is on machine-to-machine communication and encoding of units. Unlike the latter two standards, which it claims contain numerous name conflicts and are incomplete, UCUM "provides a single coding system for units that is complete, free of all ambiguities, and that assigns to each defined unit a concise semantics."

The standard is in scope and of (limited) interest to our application

7.3.2 Applications

GNU Units String-based command-line application for converting units. Available as Linux and Windows binaries.

jScience This library is, among others, an implementation of the UCUM standard mentioned prior. The library was part of a Java Specification Request (JSR) to be made part of the Java Standard Library under JSR-275, which was rejected in 2010.

The project itself is not in active development and can no longer be downloaded from their main site, as the source code and binaries were hosted in the now-defunct java.net platform.

JSR-363 – Units of Measurement API Based on jScience (JSR-275), this library provides a rich programming interface to express quantities and units in Java. The proposal is on its way to being approved at the time of writing (July, 2017).

7.3.3 Conclusions of Survey

The Units of Measurement API (JSR-363) is promising. However, given the fact that many such libraries have failed to gain noticeable traction in the past (at least in Java), and because we want to keep the learning curve for enthusiast data importers as low as possible, we consider this API to be too complex for the simple task at hand: to convert units.

7.4 Implementation

7.4.1 Converters

First, let us consider the `Serializable` Java interface. In serialization, the goal is to transform an in-memory object into a format which enables its state to be persisted, and conversely to re-create an object from this persisted state to an in-memory object again. To accomplish this, the interface requires one method to serialize the object, and one to deserialize, respectively. Additionally, because the serialization implementation may have changed between the time an object was serialized and when it will be deserialized, the interface imposes a version descriptor (simply a `static final long serialVersionUID` value).

Our task is quite similar to serialization, not in that we seek to persist an object (which has attributes with values), but rather in that there is some process which renders a particular instance of a "thing" into a different *representation*; additionally, it's also able to reverse this process. Versioning is naturally of interest as well, since without some mechanism to revert a conversion, a faulty unit converter would permanently render converted measurements unusable and irrecoverable. A converter version would enable us then to revert incorrect conversions.

At its simplest, a unit converter could be expressed as:

```
public absrtact class UnitConverter {

    public abstract double convert(double source);
    public abstract double inverse(double source);
}
```

And an example Celsius-to-Farenheit converter:

```
public class CelsiusToFarenheitUnitConverter extends UnitConverter {
    @Override
    public double convert(double source) {
        return (source * 9d)/5d + 32;
    }

    @Override
    public double inverse(double source) {
        return (source - 32) * 5d/9d;
    }
}
```

The converters would be made available to the community and open for contributions. This can be achieved through a public version control system (such as Git), on which merge requests could be accepted.

7.5 Discussion

Authoriship: Written by Paul, proofread and edited by Andres

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 ensure that all data in the database has the same unit for a given measurement type. Of course we cannot enforce that the user does indeed convert measurements correctly or at all, but this would be considered 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 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.

7.6 Future Improvements

Our implementation in this regard is a proof of concept. The community-sourcing aspect remains to be implemented, though, as discussed, a public Git repository would be a feasible low-effort first alternative.

8 Inserting into Elasticsearch

Authorship

Version	Date	Modified by	Summary of changes
0.1	2017-07-24	Paul	initial version
0.2	2017-07-30	Andres	Proofread

This section covers our consumer component, which is responsible for taking data items from the queue and inserting them into the main database. We will discuss the requirements and the tasks it fulfills.

8.1 Requirements

In our architectural design, data importers push their output to a queue, in our case *Apache Kafka*. From the queue, the data items must be inserted into Elasticsearch. Besides simply taking data from the queue there are additional validation requirements that this component fulfills.

8.2 Validation

The output generated by data importers (JSON documents) that are prepared and processed must be validated prior to insertion in the database. That task is logically linked to the data storage component and not the importing process itself since:

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

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

8.2.1 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 (channels) it uses. The consumer should work and be reconfigurable, if our Kafka configuration chances

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.3 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 separate steps with another queue in between as well. This would have meant a lot higher configuration effort and more resources required so that it seemed good to join both tasks in one component.

8.3.1 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 as well 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.

8.3.2 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 *JSON Schema* 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.

8.3.3 Allow concurrency 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 built-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.

8.3.4 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 data source (e.g. when several importers run for several days for a data source.)

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 success 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 data source 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.

9 Database

Authorship: Written by Nico Tasche
Minor proofreading & editing by Andres Ardila

9.1 Overview

The chapter is all about how the collected data is stored and the API to access the database. The focus is going to be database, the requirement, decision making, architecture and data model. The API handles mainly the access to the database and implements some optimizations mentioned in the database article.

9.2 Requirements

9.2.1 Primary Requirements

- scalable in the range of petabyte in size
- hundreds of thousands of requests per minute
- highly available
- partition tolerant
- fast to handle time-series
- fast to handle geolocation data
- immediate consistency is NOT necessary

As seen in the requirements, a huge focus was on scalability. We had some secondary requirements as well, which were mainly regarding our possibilities to handle the project.

9.2.2 Secondary Requirements

- Open source or at the very least an open license is required.
- must be well documented
- must be manageable regarding administration and learning afford

9.3 Survey of Existing Solutions

Based on our knowledge, any relational database as main data storage has been quickly disregarded. Fairly bad scaling behavior with the amount of data we should be able to handle and the fixed data-schema architecture were not a good fit for our use case. That does not mean, that a relational database might be used for some special requirements.

Based on the preexisting experience in the group we had a closer look at Elasticsearch and checked it against our requirements. This first look was very promising, but more on that in the next chapter.

Other solutions in the realm of NoSQL databases were mainly web research. That included what is out there and what is the strength and weaknesses of each solution. That included mainly the big known ones like Cassandra, MangoDB, . . .

The survey of other existing NoSQL database solutions was not very extensive, regarding real tests and in deep research. Because of the feedback, which suggested we use the wrong database, we decided in the middle of the project, to have an extra look at Apache Cassandra.

9.4 Evaluation Criteria & Decision-making Process

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

Besides the previously mentioned requirements we had some soft-requirements as well:

1. spread of database system
2. liveliness regarding
3. what was already known in the group

9.4.1 Why Elasticsearch

Checking our requirements Elasticsearch fulfilled all of them. The reasons why we decided to use Elasticsearch even before we went to deep into other databases were as follows: - native support for geo spatial searches [8] - group members already knew Elasticsearch, unlike all other NoSQL databases - extremely well documented - very high development pace from the company who develops it

Considering we had no real database expert in our team, the fact that group members already had practical experience with Elasticsearch was a very important point for us.

9.4.2 Special look at Cassandra

After a few tests and an overview of the documentation, it was clear that Cassandra had two shortcomings regarding our project. First, the documentation seems to have quite a few gaps [2, 3]. Second, there is no native support for spatial geo data.

9.5 Implementation Details

9.5.1 Intro to Elasticsearch

Elasticsearch [1] is an open source Lucene based search engine. It is under active development, with an extensive documentation.

9.5.2 Architecture

Elasticsearch is a document store and works with indices, which are comparable to tables in the classic SQL world. Each index holds JSON based document which follow a mapping. The mapping is flexible and can be extended however you want, but as soon as a field in the document has a mapping, all documents with the same field have to follow that mapping. E.g. if a document has a field called location, which is mapped to a geopoint data type. Every document with a field name location, must have a geopoint in it.

Each index can be sharded and each node/server-instance can have multiple indices. To better distribute search requests, the workload is divided among all shards belonging to an index and means that each shard holds just one part of an index. Because that would not scale very well and would have no partition tolerance, each index has a configurable number of replicas. A new search request is send to one replica of each shard.

Elasticsearch automatically handles the distribution of search requests to the primary shards or replicas. Depending on the number nodes in a Elasticsearch cluster, the engine also automatically distributes the shards and replicas to the different nodes, all according to the configuration made for each index. To illustrate that see the following figure, which assumes that the cluster holds three indices, on three nodes, with two replicas per index.

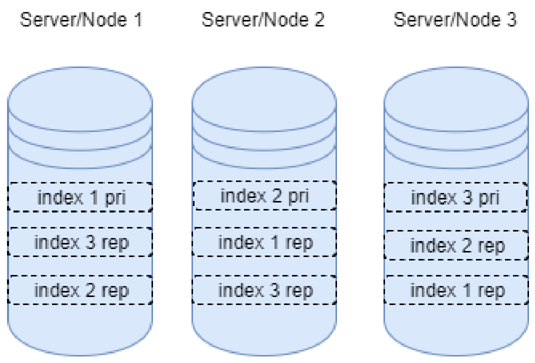


Figure 8: OpenData Database Architecture

The index configuration in our architecture follows a template, which is automatically applied to each new index we create:

```
{
  "template": "data-*",
  "order": 1,
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 3
},
"mappings": {
  "_default_": {
    "_all": {
      "enabled": false
    }
  },
  "data": {
    "properties": {
      "device": {
        "type": "keyword"
      },
      "location": {
        "type": "geo_point"
      },
      "timestamp": {
        "type": "date"
      },
      "timestamp_record": {
        "type": "date"
      },
      "license": {
        "type": "text"
      }
    }
  }
}
```

This template provides the number of shard, replicas and the basic mapping for our data model. If necessary this information can be overwritten before a new index is created. This is all the necessary information needed for our architecture and cannot be changed after an index is created, except for the number of replicas, which can be increased.

9.5.3 Data Model

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

Those are the only information we need to store, besides the individual measurements. We do actually store some more information, but regarding the common use cases for searches those two data points are enough for environmental data. For each data entry then, there are one ore many measurements in that data point. Each measurement than has a quality indicator, a observed value and an sensor name. Please refer to the full data-model in the project wiki for more information.

There were mainly two reasons to use this data model. For one it keeps the data provenance, which we quite like to keep. The second reason is that it is nice to handle in terms of partitioning.

9.5.4 Partitioning

As for the question, how does the used data model scale and how to best partition the imported data we decided for a 2-dimensional approach. One dimension is already covert by the data-source centric data model, which allows us, to partitioning by source. The second dimension is the time. Each data source is partitioned based on the time a measurement has been taken and the granularity can be adjusted as well.

This gives us multiple advantages:

- it allows us to adjust the server infrastructure based on the data source
- it scales indefinitely
- index size is deterministic, because 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 transferred to another Elasticsearch node with different hardware. That would be useful, for example, when the average density of the smurf population is being stored. The index can be transferred to a less powerful hardware with fewer CPU cores and spinning hard drives and even fewer replicas, because this information is probably hardly requested, except from Gargamel and maybe some surf protection groups.

As a starting point, we choose to configure each index as shown in the figure below. We decided to keep each index on one shard, which lowers the network traffic and because we are very flexible regarding the size of the index thanks to our time-based partitioning, this should not become a problem later on. To allow thousands of request per second, we decided to start with three replicas, so each request is forwarded to a different replica. This can be flexible in- or decreased later on.

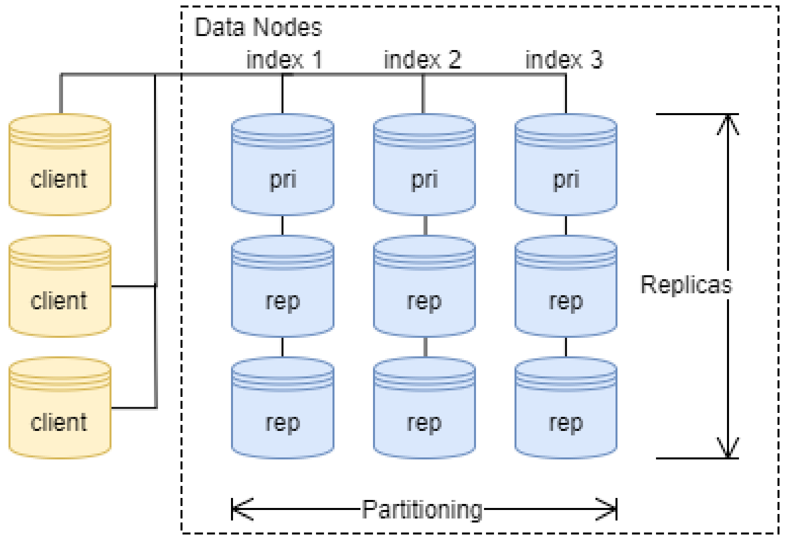


Figure 9: OpenData Database Architecture

9.5.5 Query Optimization

Why do we need query optimization? For that I'm going to give a small example to consider:

1. we import multiple sources, with multiple measurements: source1(air-temperature, water-temperature) 1980 - 2017, source2(air-temperature) 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 naively sharded over 3 nodes (we actually use just one shard per index)

Let's make a simple search request: give me all UV values data from 2015 till 2017 and aggregate an average over the month. Because the user does not know 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 indices, 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 indices 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 already reduced the number of shards we need to address by 61%

With a little more sophisticated time limitation algorithm, we could actually 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 shards
source3 = 6 shards

source1 + source2 + source3 = 81 shards
```

Now we are at 68% reduction.

Second optimization: Limit to indices which contain the right data

If we store in a separate database, which data source and therefore indices actually hold the requested data we can do even much more:

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

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

source1 + source2 + source3 = 6 shards
```

By using those two optimizations, we could reduce the number of requested shard to 6, which means a total reduction of 96.2%.

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

9.6 API Implementation Details

9.6.1 Overview

As for the API, it was important for us to provide a solution which is easy to scale via a load balancing, for example with an nginx instance as an entry point for the user of our system. That means, those instances need to be stateless. For that and because it is basically a standard we use a REST interface to provide access to our data-collection.

9.6.2 Architecture and Technology

As base technology, we use NodeJS, which is easy to use JavaScript runtime based on Chrome's V8 JavaScript engine. It allows a fast iteration pace and needs just minimal preparation to develop server instances with.

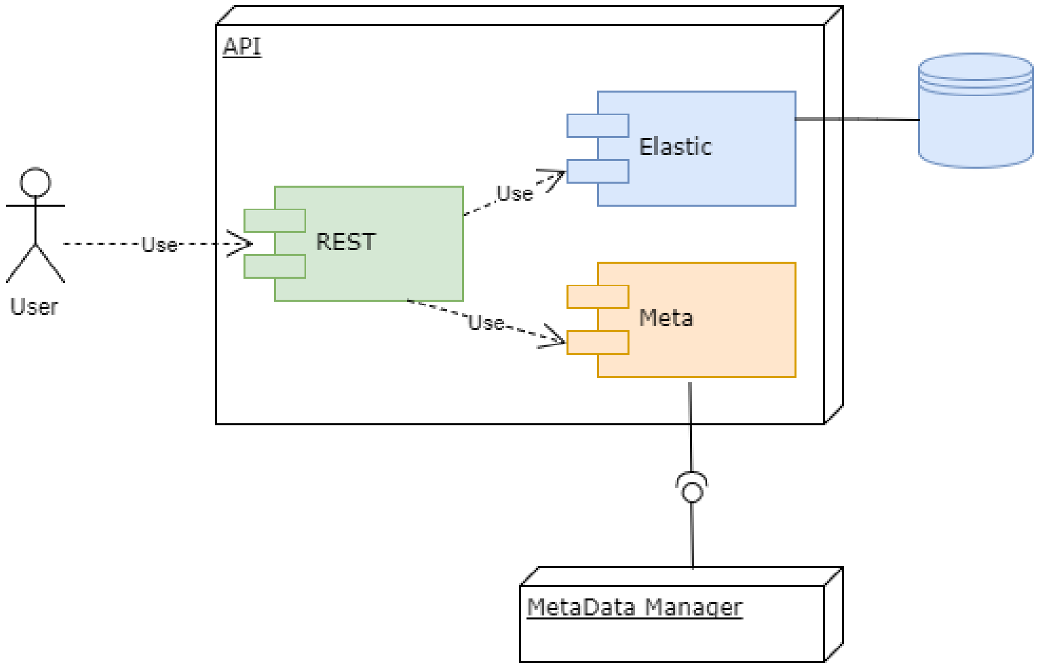


Figure 10: Public API Abstraction

The API implementation consists of three parts.

1. Route: defines all routes and parses all parameters to be used in our system
2. Meta: connects to the management database and requests all necessary data (this part is prepared, but connection not implemented yet)

3. elastic: does everything Elasticsearch specific and could be replaced by another database connection if the database would be replaced for example.

9.6.3 REST APIs

The REST API currently consists of three endpoints:

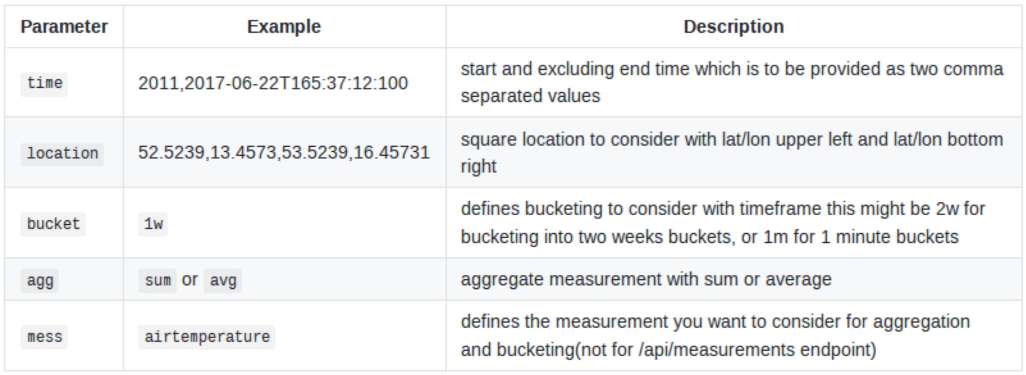
1. GET /api/sources/
Returns all currently available resources

2. GET /api/sources/:indexName
Returns data based on the sources they came from

3. GET /api/measurements/:measurements
Returns data based on the measurements you're requesting

The 2nd as well as the 3rd REST-endpoint allow a more specific search, based on extra parameters.

Parameters:



Parameter	Example	Description
`time`	2011,2017-06-22T165:37:12:100	start and excluding end time which is to be provided as two comma separated values
`location`	52.5239,13.4573,53.5239,16.45731	square location to consider with lat/lon upper left and lat/lon bottom right
`bucket`	`1w`	defines bucketing to consider with timeframe this might be 2w for bucketing into two weeks buckets, or 1m for 1 minute buckets
`agg`	`sum` or `avg`	aggregate measurement with sum or average
`mess`	`airtemperature`	defines the measurement you want to consider for aggregation and bucketing(not for /api/measurements endpoint)

9.7 Discussion

9.7.1 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 must be done either on a separate server, like our API instance, or on the application side. This is something we were not aware of for quite a long time.

9.7.2 Administration

This is probably not a Elasticsearch specific but should be mentioned in this chapter as well. To setup this the basic database is quite easy, but to scale it to up to petabyte needs quite a bit consideration. Our data model and the current implemented optimizations will help scale the database and everything should work without any performance drawbacks for quite a while. To archive this, lots of working though documentation and local request testing had to be done. As said before, this is probably for every other database as well.

9.7.3 Testing

Testing is something we could just do on a very limited scale. Unfortunately, any possible short-comings in our architecture would just show much later then we could test.

9.8 Future Development and Enhancements

9.8.1 Data Postprocessing

As mentioned in the Limitations section, joins are not possible in our system right now. One way of doing it would be to allow application based or backend based join, which will have same limitations. A better way would be to post process our data at low usage times. One idea would be to collect all the data for a defined area and put all information we have about that area into one dataset. The area could be for example a size of 100m x 100m. This would allow for very fast, very complex queries, which involves quite a few measurements.

9.8.2 The API

The current implementation of the API is somewhat limited. This is mainly due to time limitations while implementing. These are the points, which should be implemented:

1. Allowing to make post requests to make more complex queries. That would for example include the transfer of big geospatial shapes for filtering.
2. Adding security with API tokens
3. Adding full access to the management database
4. Adding safety checks like timeouts to make not to extensive requests
5. Adding possibility for scrolling to request data in chunks
6. optimizing the data handling on the API instances, which is not very efficient

10 Data Source Metadata Management System

Authorship:
Written by Paul Wille
Proofread & edited by Andres Ardila

In this chapter we will discuss the component responsible for managing metadata about the data sources which are imported to the system. We will cover what it is and does, some thoughts around why we decided to include such a component, and how it was realized and implemented.

10.1 Requirements

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

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. However, the following general cloud-specific architectural style requirements were taken into consideration:

- 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)

10.2 Implementation

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

10.3 Domain Model

10.3.1 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

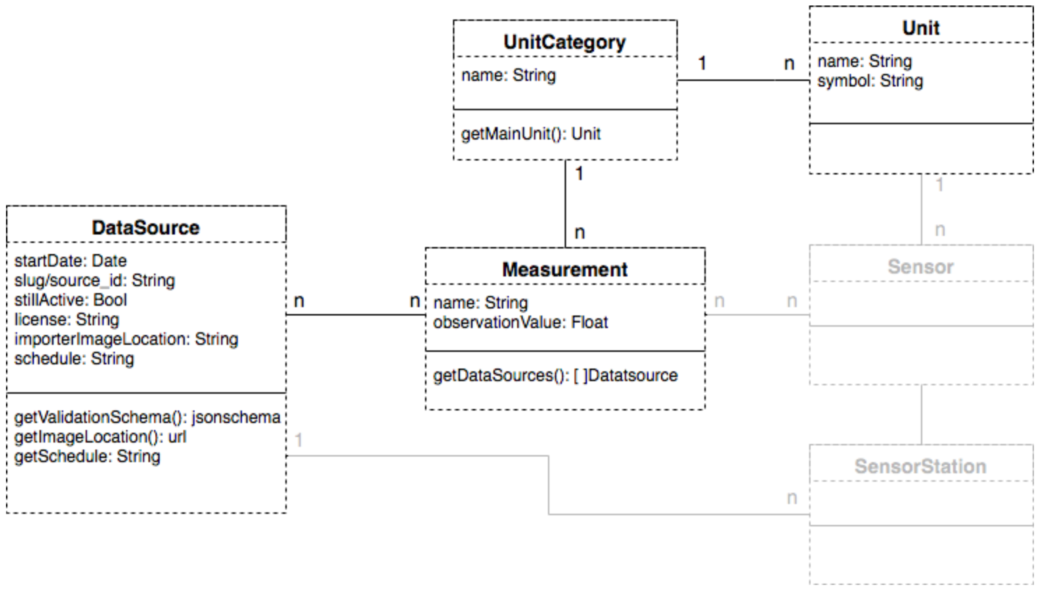


Figure 11: UML Class Diagram of the Metadata Management System

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

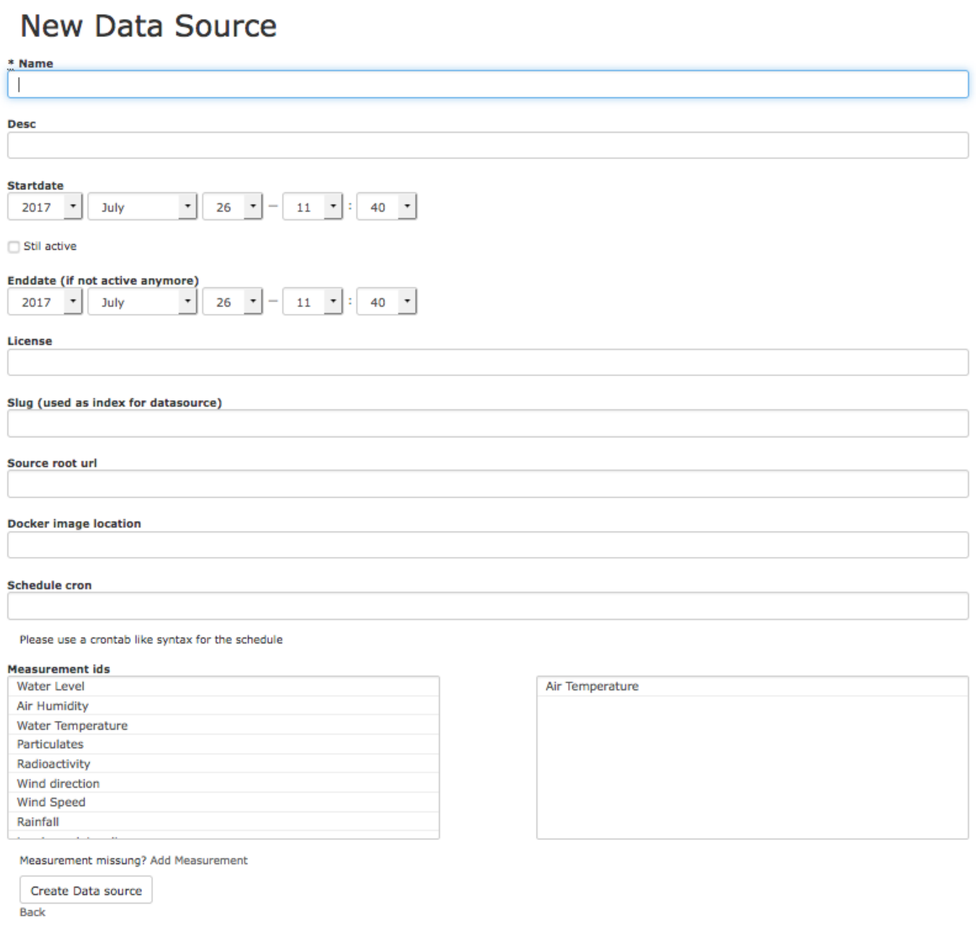
Additionally we ask the user to provide additional information, which is useful for our system:

- 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.

10.3.2 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:


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

Measurement missung? Add Measurement

Create Data source

Back

Figure 12: Registering a data source within the Web management platform

- 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": "An open sensor data source",
  "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"]
}
```

10.3.3 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 (see Figure 12).

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 can be seen in the class diagram 11 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_category_id":"humidity"
   }
]
```

```
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":"" }
]
```

10.3.4 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.

10.3.5 Units

With the requirements in mind, we modeled units like so:

1. **Unit Categories**: Units themselves 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 11).
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 straightforward. 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 (Figure 13).

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
```

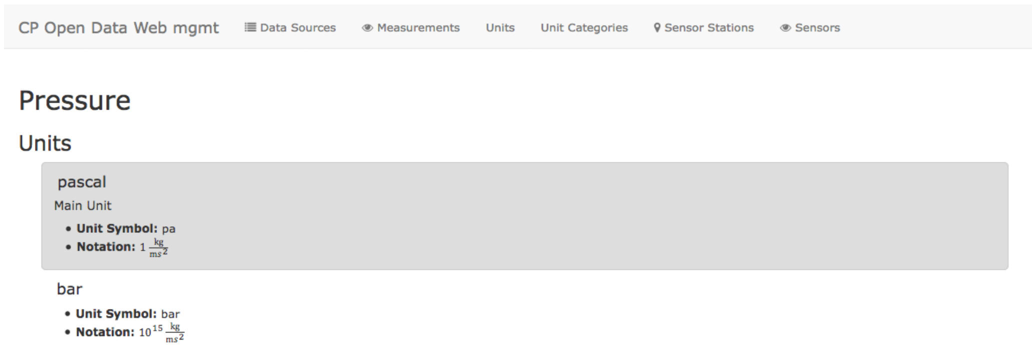
Example request:

```
GET /unit_categories/temperature/getMainUnit

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

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

Example request:



Pressure

Units

pascal

Main Unit

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

bar

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

Figure 13: 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.

```
GET /unit_categories/temperature/units

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

10.4 Discussion

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 the UML Class Diagram 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)

10.4.1 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, 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.

11 Conclusions

Authorship

Version	Date	Modified by	Summary of changes
0.1	2017-07-19	Oliver, Andres	Working draft
0.5	2017-07-30	Oliver	From bullets to text

11.1 Is Open Data really open?

The main feature of Open Data is to be easily available and accessible to everyone. During our research for sources to include and collect data, one of our main findings was the difficulty in finding the specific type of non-static timeseries environmental data in which we were interested. A large portion of data "sources" present their data to the user through a GUI, but do not provide the underlying data. Some providers even prohibit the usage and storage of the data for commercial and even private usage outside their platform. Furthermore, the overwhelming amount of disparate protocols to access data in different formats (CSV, XLS, HTML) in which the data is presented and offered makes it hard for the user to benefit from the data.

Still, we were able to identify a multitude of available sources and used our system as proof-of-concept to include and provide seven of them (see the Appendix for a list). The "'unlocked" sources offer different measurands like temperature and river water level.

Though, concluding one must say that nonetheless, a lot of work is to be done when it comes to make Open Data really **open**.

11.2 Lessons learned

The best way to improve is to learn from mistakes, should they be your own or someone else's. But first you must be aware of challenges and setbacks along the way. To do so, this section will recapitulate the history of the project and discuss the lessons learned.

Project teams will always consist of heterogenous people with different skills and understanding of the problem. To get everyone on the same page is a difficult challenge and with such strict time restrictions, it got even more challenging. Cultural and language proficiency differences often prevented us from reaching consensus fast and function as a unit. Furthermore, team members tended to work separately. This in itself is not generally a problem, but in our case, due to lack of communication and 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.

Because of the lack of cloud infrastructure, testing became a problem, which led to bugs or issues in the individual components not being discovered until late in the process, more specifically until deployment of the whole system. Further, because the infrastructure deployment basically rested on the shoulders of a single member, this created a bottleneck and excessive load on this person.

Lastly, the tools which were there to keep order and improve team work and productivity got partially abandoned. Specifically, somewhere around the midpoint of the project timeline, most team meambers stopped using Trello to keep track of global progress and open 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. The latter was in part a consequence of the difficulties in finding consensus; as individuals found it difficult to work together, they isolated themselves more and more from group work.

Maslow's hammer:
"when you have a hammer, everything looks like a nail!"

Moreover, we experienced Maslow's hammer first hand. In our case this means that we tried to adjust the problem to our technology rather than vice versa. Too often, the implementation depended on the developers prior knowledge of a given technology, and not as a result of a contentious and reasoned analysis of the technologies and tools available, a comparison of what characteristics they exhibit under a specific set of criteria based on desired results or use cases, etc.

Concluding, a huge lesson that we learned is to start with a use case and then try to find the best solution. For example, when choosing a database technology start with the queries you want to optimize and for which you want to ensure the perfect performance. No system can be the best solution for every given challenge thus, formulate your problem statement specifically and based on this, choose the best fit. Moreover, when the decision for a technology or solution was made, test and benchmark it under real-world conditions rather than conjecture. Though, said approach is hard to implement given the limited time and budget we were given. A different trade-off that has to be made is the time invested in learning new technologies against exploiting existing knowledge and understanding. Typically, in a real-world case a project team is formed of domain experts so that existing knowledge can be harnessed most of the time. In our very unique case, a team formed of diverse university students, the available knowledge and experience with existing solutions might not suffice. Hence, we were presented with problem domains to which we did not have the right solution and expert. This forced us to spend a significant amount of time to research available technologies and services.

Finally, never underestimate the overhead of cloud resource procurement and the administration of the infrastructure. Obviously, without available infrastructure testing and benchmarking is not possible thus, decide early in the project where and how to test. It might not be the optimal solution but as the product grows everything can improve.

References

[1] Elasticsearch homepage.

[2] Missing documentation in cassandra.

[3] Missing documentation in cassandra 2.

[4] Len Bass, Paul C. Clements, and Rick Kazmann. Software architecture in practice. 2012.

[5] Andreas Chatzakis. Architecting for the cloud: Aws best practices. 2016.

[6] David Garlan and Mary Shaw. An introduction to software architecture. 1994.

[7] Khtan66 (Own work) [CC BY-SA 4.0], via Wikimedia Commons. Kubernetes, 2016. [Online; accessed July 28, 2017].

[8] Michael McCandless. Multi-dimensional points, coming in apache lucene 6.0, 2016.

[9] Eberhard Wolff. *Microservices: Grundlagen flexibler Softwarearchitekturen.* dpunkt.verlag, Heidelberg, 1., korrigierter nachdruck edition, 2016.

A Imported Data Sources

No.	Name	Measurand(s)	Data Format	Author
1	Weather Data Stuttgart	Humidity, Temperature, Pressure	CSV	Jawid/Rohulla
2	Luftdaten Brandenburg	O_3, NO, NO_2, PM10, PM2.5, SO_2, CO	XLS	Jawid/Rohulla
3	Umweltbundesamt	PM10, SO_2, O_3, NO_2, CO	CSV	Jawid/Rohulla
4	Pegel Online Water Level	Water Level	REST, SOAP	Jawid/Rohulla
5	BLUME	N_2O, SO_2, PM10, C_6H_6, CO, O_3	HTML	Andres
6	BFS UV Index	UV Index	HTML	Oliver
7	NOAA GHCN	Weather	DSV	Andres

B How to Write a Data Importer

Authorship

Version	Date	Modified by	Summary of changes
0.1	2017-07-29	Jawid, Rohullah	Initial version

This chapter guides you through building your own importer on top of an already existing template.

B.1 Template

Instead of wrting a long manual on how to write an importer we provide you with starter apps (templates) that you can start to customize right away.

B.2 Pre-requisites

We assume that you are:

- comfortable working with Java and Spring Boot
- already familiar working with Spring Batch
- having knowledge of working with multi module applications using Maven and Spring Boot.

If you need a quick start guide of Spring Batch, you can find one at `https://projects.spring.io/spring-batch/`.

B.3 How to Proceed

1. Download or copy the template that fits your needs.

Template Modules
The template is a multi module application which includes two modules:

- application
- library

These modules are defined inside `pom.xml` file.

```
<modules>
    <module>library</module>
    <module>application</module>
</modules>
```

2. Start by changing the importer name inside pom.xml in the root directory of the project. You don't need to change anything else inside pom.xml.

```
<groupId>de.tu_berlin.ise.open_data</groupId>
  <artifactId>your-importer-name-here</artifactId>
  <version>0.1.0</version>
  <packaging>pom</packaging>
  ...
```

3. Once you finished editing `pom.xml` you can open it as a project using your IDE. Now you can start to customize it.
4. Before changing anything else you can see how the importer works by running a kafka server locally and starting the importer's main class using your IDE.

   OR

   Inside the root directory of the importer run these commands:

   ```
   $ cd application
   $ mvn spring-boot:run
   ```

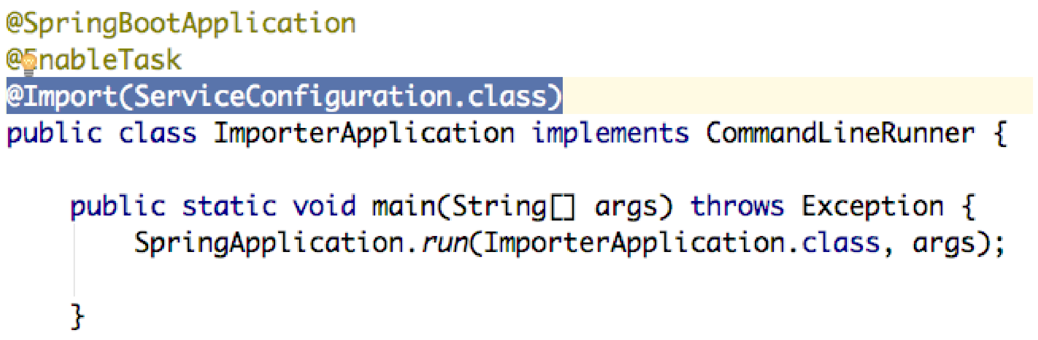
5. You can now see inside the console how a simple job runs.

B.3.1 Things to know about the template:

- Inside the `pom.xml` in the 'application' module the following dependency is used to include the 'library' module:

```
<dependency>
  <groupId>de.tu_berlin.ise.open_data.library</groupId>
  <artifactId>library</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

- Registered beans inside the library module could be autowired by importing the `ServiceConfiguration` class from package `de.tu_berlin.ise.open_data.library`. You can see how it is imported in main class:



```
@SpringBootApplication
@EnableTask
@Import(ServiceConfiguration.class)
public class ImporterApplication implements CommandLineRunner {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(ImporterApplication.class, args);

    }
```

You can now start to edit the template.