Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Integration of standard datasources with interactive data visualization solutions

Bachelor's Thesis

Author
Márton Orova

Advisor
dr. Zoltán Szatmári
Attila Simon

December 5, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Orova Márton*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 5.

Orova Márton
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Chapter 1

Introduction

Nowadays, the question of storing, processing and displaying the data is becoming more and more important throughout every industry. The time, when the collected data was only useful for computers and specialists, passed. Today, the need for showing data in an easily understandable form is significant. It is no wonder, people through the whole hierarchy of a company would like to be well-informed about the results and the ongoing processes. In addition, it is getting highly valuable to be able to display vast amount of data in a way that even outsiders can comprehend.

Because of this trend, many technologies attempting to solve these problems have appeared, creating a wide variety of tools which organizations can use.

In enterprise-grade environments, the use of complex systems - so called data-pipelines - are becoming increasingly common. These tools provide an integrated solution for transforming and querying data coming from data sources built with different technologies. With the help of these data-pipelines, it is possible to collect many types of data, no matter the format or the frequency.

All these things for one reason, to prepare the data for machine or human decision-making.

1.1 Problem definition

Every organization needs to manage the sometimes cumbersome task of managing data. It is common, that this data does not come from one place, but from multiple sources, which can present various problems, especially if at some point, merging all the available data is necessary. It can happen for example, if the organization wants to visualization of the data to be in one place.

The most fundamental one is maybe the issue of the data format and the way of accessing the data. There are already numerous industrial standards available, however, using these together can cause difficulties. The problem is not only the conversion between different data formats, but also handling the different frequency of incoming information from each data source.

In addition, each of these heterogeneous components need to be configured in different ways, with varying set of parameters. Keeping up with all of the technologies, which may change over time independently from each other. This can produce a heavy overhead for the data-specialists.

Thus, we can establish the notion, that an universal tool for integrating different types of data sources is highly desired to allow the visualization of the recorded information in one place.

1.2 Motivation

There are numerous reasons why I choose to work on this thesis project. In this section I try to collect them together in order to better express my motivation towards this task.

It was expressed in the previous section, that it is quite valuable for an organization to be able to display its collected data in one visualization tool. Achieving this would allow an easily understandable and centralized way of presenting the data.

Thorough this project, I focus on integrating different data sources with a popular, open source visualization tool called Grafana. Working with this allows me to get familiar with the details of complex systems and their architecture. To combine the data sources with Grafana, I need to look into many different technologies, which surely widens my knowledge about various programming languages and design principles.

1.3 Goals

There are a couple objectives to achieve during this thesis project.

First of all, an investigation is needed about the mainly available data sources and their varying features. This is necessary to get into context and to establish further decisions concerning this work.

Following that, we have to conduct a research on the available interactivity capabilities of Grafana. It would be great, if thorough this project, we could utilize as many of them as possible and integrate them into our work. In addition, as Grafana is an open source project, there are probably plenty of possibilities for extending or customizing it in the context of interactivity. We should briefly look into it and make an attempt at implementing some simpler use-cases.

As it was already expressed above, softwares for integrating different types of data sources are indispensable. We need to design a tool that can handle this task in case of two data sources and connect them to Grafana, an industry standard for data visualization. More specifically, we need this component to be able to connect a RapidMiner data source and a Python data source to Grafana in way that allows a two-way link between them to enable further interactivity capabilities.

After the design of such tool, a sample implementation is needed to ensure the functionality of the architecture and demonstrate its effectiveness. Afterwards, we need to analyze the created gateway, discovering and presenting its advantages and disadvantages.

Chapter 2

Background

2.1 Data sources

A data source is what its name suggests, a location where the data that is being used originates from. A data source can be for example a database or a dynamic stream of data service such as constantly collecting the measured temperature in a room. In the context of this thesis project, we focus on database-like data sources, because in general, a visualization tool needs to be able to read the data to be displayed from a backend storage.

Concerning the types of databases, there are different categories, each with varying features and purposes. In the following sections, I briefly introduce some of the most used types of databases.

2.2 Relational database

Probably the most popular type of databases is the relational database. It stores data points that are related to one another. Relational databases are based on a relational model, representing the data in tables. They are widely utilized in cases, when related data points need to be stored and managed in a secure, rules-based, consistent way.

2.3 Time series database

A time series database is optimized for storing time-stamped or time series data. Time series data are measurements that are tracked and aggregated over time. This

could be metrics about servers, application performance, network data, etc. A time series database possesses enhanced capabilities to handle the measurement of change over time compared to other types of databases.

2.4 Key-Value database

A key-value database is type of nonrelational database that uses a simple key-value method to store data. The data is stored as a collection of key-value pairs in which the key fulfills the role of a unique identifier. Both the keys and the values can have any type, simple objects, like number or string or even complex compound objects. This means that the schema of the data is defined per item, instead of on a collection or table level, like in the case of relational databases. Key-value databases are mostly utilized for cache management or storing information in session-oriented applications.

2.5 Document database

Document databases are another type of nonrelational database that is designed to store and query the recorded information as flexible, semistructured JSON-like documents. They make it easier for developers to handle data in the database by using the same document-model format they use in their application code. Document databases allow flexible indexing and powerful ad hoc queries.

2.6 Grafana

Grafana is one of the most popular open source analytics and monitoring solution that can be connected to the majority of the main data sources out-of-the-box. It allows its users to query, visualize and alert on the collected metrics.

Altough Grafana has got plenty of useful feature, only those relevant to the scope of the thesis project will be briefly explained here

2.6.1 Data Source

Grafana supports many different storage backends (Data Source). Each Data Source has a specific Query Editor (see later) that is customized for the features and capa-

bilities that the particular Data Source exposes. Of course, this leads to the fact that the query language and capacity of each Data Source are obviously very different.

Grafana mainly favors time series data (e.g. from Prometheus or InfluxDB), but it can work with other types of data source (e.g. relational databases (MySQL, PostgreSQL, MSSQL), logging and document databases (Loki, Elasticsearch)).

2.6.2 Panel

The Panel is the basic visualization building block in Grafana. Each Panel provides a so called Query Editor (dependent on the Data Source selected in the panel) that allows the user to create data source specific queries (e.g. an SQL query for a MySQL data source) in order to extract the desired metrics as precisely as possible.

There are multiple built-in Panel types available in Grafana, however, custom panels made by the open source community are also accessible. Probably the most widely used Panel types are the Graph, Table, Singlestat and Gauge. An example of the Graph Panel can be viewed in Figure 2.1.

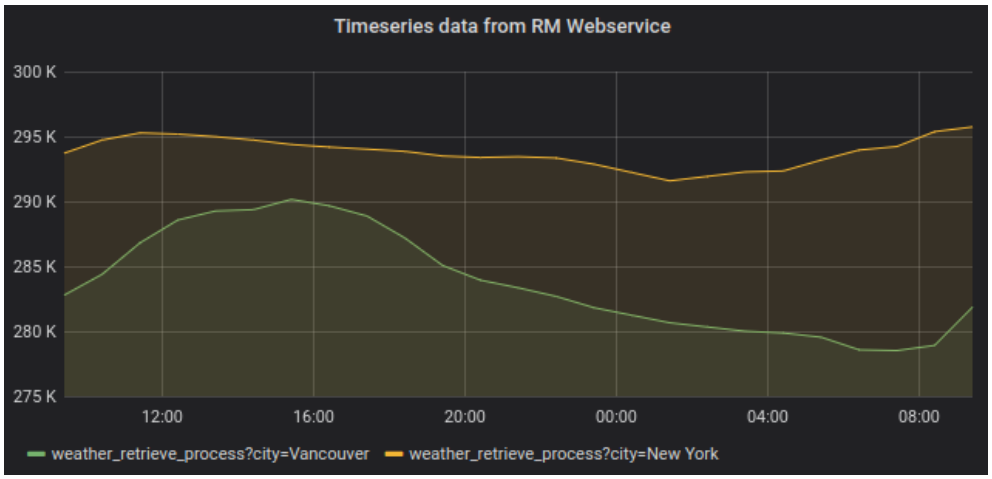

Figure 2.1: Graph Panel

2.6.3 Query Editor

The Query Editor exposes the capabilities of the Data Source and allows the user to query the metrics that it contains. The queries created in the Query Editor of a panel determine what data will be displayed on the panel. An example Query Editor is shown in Figure 2.2.

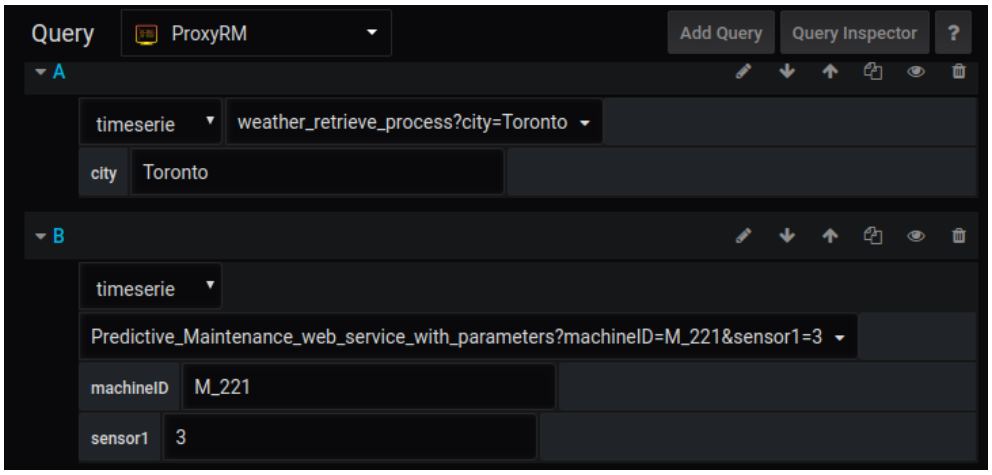

Figure 2.2: Query Editor

2.6.4 Dashboard

The Dashboard is where all the previously mentioned building blocks come together. Dashboards can be thought of as an organized set of Panels.

We can use the Dashboard to visualize different metrics in one, easily manageable place. This is quite useful, when many aspects must be taken into account in order to be able to thoroughly understand our currently inspected data set. An example Dashboard about a Kubernetes cluster can be viewed in Figure 2.3.

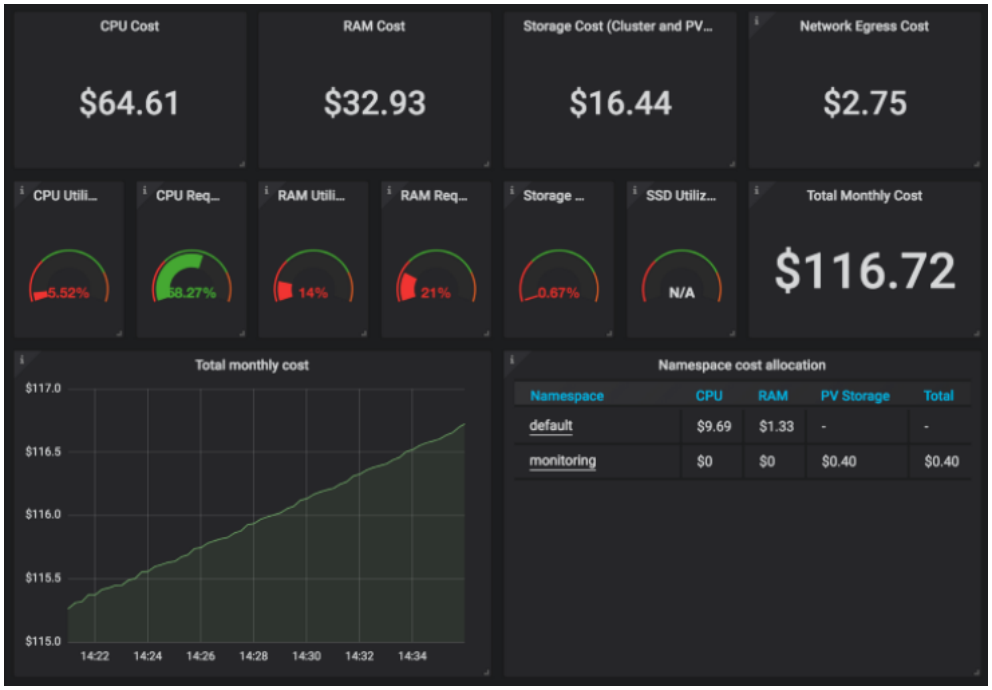

Figure 2.3: Dashboard https://grafana.com/grafana/dashboards/8670

2.6.5 Interactivity in general

When working with and visualizing massive amount of data is a major part of one's profession, it can be quite convenient if the tools can provide the ability to allow some user-interaction. Meaning that the user do not have to work with static diagrams or rewrite complex configurations so that he or she is capable of further analyzing the collected data.

Commonly available interactivity options can be the followings:

- *statistic indicators* - easily accessible statistic information (e.g. average value, spread-diagram) about individual objects shown on the diagram (this is usually achieved by mouse-hovering)
- *local interactions* - changing the outlook of a diagram without affecting other displayed objects (e.g. reordering the bars on a bar-chart, rescaling the axis on a graph or other diagram-specific modifications)
- *selection and linked highlighting* - when we have multiple diagrams displaying different views of the same data set, selecting a subset of the data on one diagram (e.g. a bar on a bar-chart) highlights that particular part of the data on another diagrama (e.g. a set of points on a scatter-plot)
- *linked analysis*- for example, selecting a subset of the data triggers a reactive analysis, creating a statistic model (regression, scatter-plot, spread-diagram) using that specific data set

2.6.6 Interactivity in Grafana

While it is hard to find a tool, which possesses all the before-mentioned interactivity capabilities, Grafana provides certain features to enable efficient user-interaction for the good of solid understanding of the data.

2.6.6.1 Time Range Controls

Grafana provides numerous ways to interactively manage the time ranges of the data being visualized.

On the Dashboard-level, the 'Current time range' selector can be used to change the dashboard time. Doing this refreshes the whole dashboard with each panel on it. Those panels, which display time-dependent information, will only show data-points which timestamps are in the newly set time range.

If there is a Graph Panel on the dashboard, there is another possibility to change the Dashboard-level time range. We can select a time range on the Graph Panel with the cursor. This method is quite effective in case, when we would like to swiftly zoom into our data to review more refined details. An example for that feature is displayed in Figure 2.4.

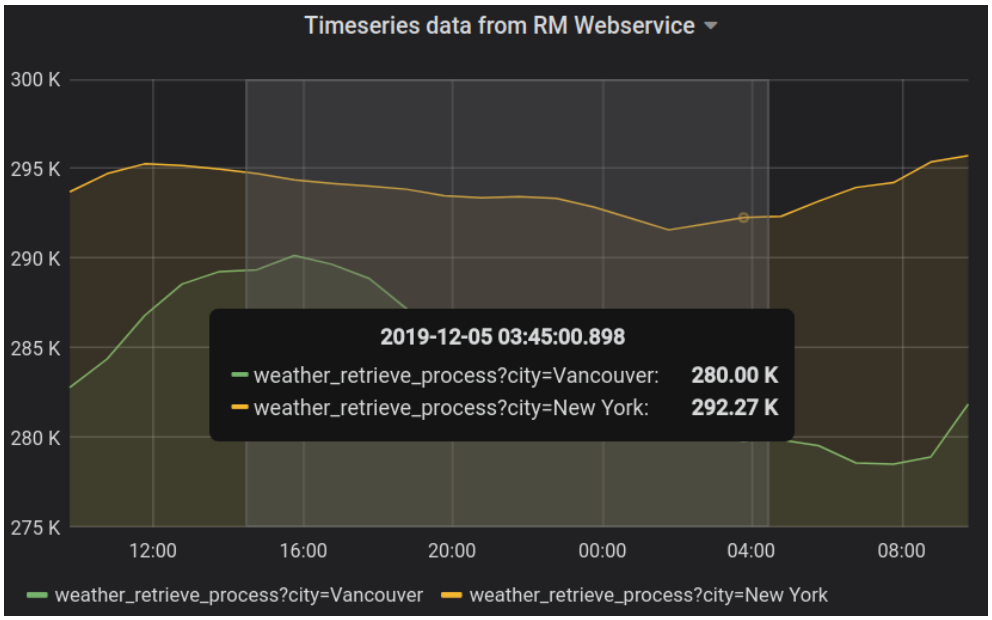

Figure 2.4: Zoom into the time range

2.6.6.2 Variables and Templating

Variables in Grafana allow for more interactive and dynamic dashboards. We can use variables in metric queries and in panel titles.

Their role is similar to that of 'conventional' variables in programming languages. Instead of hard-coding things into the program, we can use them to create algorithms, that operate on a more abstract level, resulting in more effortlessly maintainable softwares.

In Grafana, we can use the variables to write generalized metric queries. For example, if we have multiple servers and we have metric data from all of them, we could store the name or address of the servers in a variable, rather than defining numerous queries, which only differ in the server name. Variables are shown as dropdown select boxes at the top of the dashboard. These dropdowns make it easy to change the value of each available variable, thus to adjust the data being displayed in the dashboard. An example of that can bee seen in Figure 2.5.

Figure 2.5: Variable dropdown https://grafana.com/static/img/docs/v50/variables_dashboard.png

There are different types of variable that can be used to make dashboards more dynamic.

- *Query* - This variable type allows for writing a data source query that usually returns a list of metric names. For example, a query that returns a list of server names, sensor ids, or data centers.
- *Interval* - This variable can represent time spans. We can use them, when we need to aggregate data-points by time or date.
- *Data source* - This type makes it possible to quickly change the data source for an entire Dashboard. This feature can be quite suitable, when we have multiple instances of a data source in for example different environments.
- *Custom* - This variable can be used to manually define the value options for the given variable.
- *Constant* - This variable can be handy for example to easily declare metric path prefixes.
- *Text box* - This variable type displays as a free text input field with an optional default value.

- *Ad hoc filters* - It is a quite special kind of variable that only works with some data sources (e.g. InfluxDB, Elasticsearch). It allows the user to add key/value filters that will automatically be added to all metric queries the use the specified data source.

2.6.6.3 Basic statistic values in legend

Statistic indicators were already mentioned above, where we discussed interactivity in general. In Grafana, we can display some basic statistic values (minimum, maximum, average, total) in the Graph Panel. An illustration of this feature is displayed in Figure 2.6.

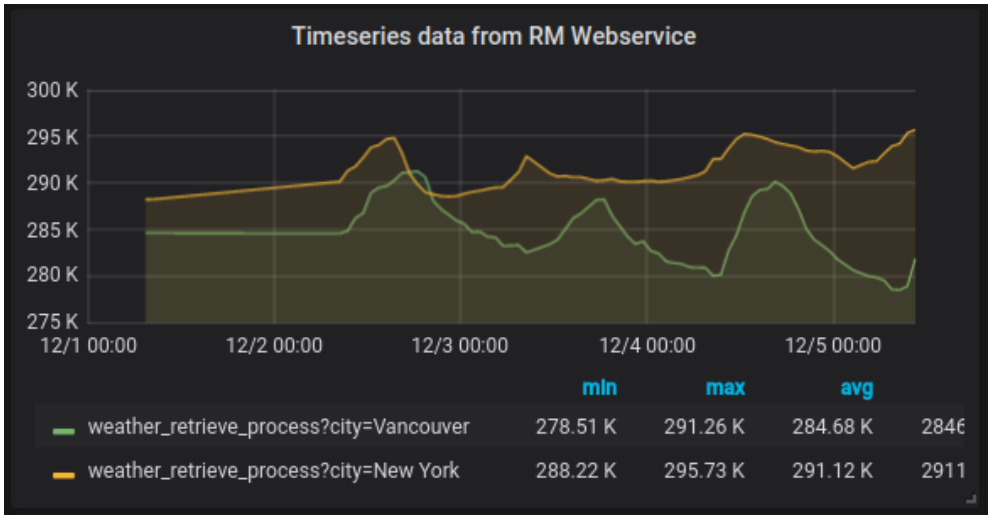

Figure 2.6: Basic statistic values on Graph Panel

2.6.6.4 Data links

Data links provide a way to add dynamic links to the visualization. These links can link to either other dashboards or to an external URL. when using data links, additional built-in variables become available that enable creating dynamic links.

These variables are:

- *___series_name* - the name of the series or the table
- *___value_time* - the timestamp of the point that is clicked on (in millisecond epoch)
- *___url_time_range* - The current time range

- *___all_variables* - Adds all current variables and their current values to the URL

2.6.6.5 Creating custom interactivity

As Grafana is a fully open source project, it is possible for the community, to develop custom objects in Grafana or to extend built-in ones. This way, it is achievable to create custom interactivity features according to the needs of the user.

As a part of my thesis project, I successfully extended the built-in Graph Panel with some time-dependency related interactivity that is explained in detail in section ..... .

2.7 RapidMiner

RapidMiner is a data science software platform that provides an integrated environment for data preparation, machine learning, deep learning, text mining and predictive analytics.

RapidMiner includes several components, however, only the ones with relevant features to this thesis project are introduced briefly in this section.

2.7.1 RapidMiner Server

RapidMiner Server is the central component in the architecture. Users can interact with it via a web interface or via RapidMiner Studio.

Its main responsibilities are to store and expose data science processes, like model building, predictions, data ETL (Extract, Transform, Load) operations, etc. Long-running jobs are executed externally via Job Agents. The RapidMiner Server stores its data in an external database.

2.7.1.1 Web services

The Server can expose certain processes that can be invoked via simple HTTP requests. These are the so called web services which can be used to run predictions or any other application of models, where the need for a real-time response is paramount. In order to make this service more dynamic, additional parameters can

be sent with the requests to get more accurate information. The user can acquire the results of these processes through the same API in JSON format.

2.7.2 Job Agent

RapidMiner Server offers a queue system for long-running jobs, which are executed externally via Job Agents. The computing power of the stack can be increased by adding more Job Agents.

2.8 JSON

In this thesis project, I use the JSON data-interchange format to send information between the various components. For this reason, a brief description of the JSON format is appropriate in this chapter.

The JSON stands for Javascript Object Notation and it is a lightweight format for storing and transporting data. We can say that JSON is easily read by both humans and machines, as a JSON object is merely an unordered set of name/value pairs. The name is a string and the values can be different types of data, for example a number, a string, an array or another object. These types are supported in all modern programming languages in one form or another, thus despite what its name suggest, the JSON format can be regarded as an universal data structure. An example JSON object can be seen in the code snippet Listing 2.1.

```
{
  "name": "Test",
  "age": 23,
  "car": {
    "type": "van"
  },
  "languages": ["english", "french"]
}

```

Listing 2.1: Example JSON

Chapter 3

Case study

It is certainly evident by now that this thesis is rather a practical implementation and an integration problem, than a research project. As one of the main goals is to combine different kind of data sources, it seems to be convenient to have some data to work with from several aspects. It makes the development easier, because we can see immediately, if the components can handle the data or that in what way they change it. The possession of sample inputs can also help to demonstrate the results and provides the capability of some independent verification.

I used the Historical Hourly Weather data set from Kaggle.com that contains hourly measurement results of various weather attributes, such as temperature, humidity, air pressure, etc. The data is available from several cities from the USA, Canada and Israel from October 2012 to November 2017.

— TODO insert extract from the data set

3.1 RapidMiner source

Concerning the part of the RapidMiner Server, it exposes a process that provides the temperature data from the sample data set. It accepts a parameter determining from which city the user wants to acquire this information. In short, it presents a filtering option by city.

3.2 Python source

In this project I demonstrate a proof-of-concept use-case, when the Python data source component reads information from a database and also from an external API

service. To be more precise, in the database I store the measured daily minimum and maximum temperature data from 2012 to 2017 in New York. Concerning the the API part, I created a small web application with a REST API that exposes historical highest and lowest temperature values for a given day of the year, also in New York. In the middleware written in Python, I implemented an example business logic which counts on how many days in a month (from October 2012 to November 2017) the measured temperature got close to the all-time records. I have to mention, that the calculation of these historical highest and lowest values is also heuristic. I only took data from January 1959 to September 2012 into account.

I believe, that this is a quite common problem, to have some data in one's storage, but for a better business competence, the usage of external services is also necessary. Hence, this example set-up could serve as an applicable demonstration for further possibilities.

Chapter 4

Design

4.1 Architecture

The designed architecture of the project can be observed in the figure 4.1. There are three data sources connected to Grafana through an extended version of the official SimpleJSON data source plugin:

- RapidMiner Server
- JSON backend
- Python data source

Each data source accesses the data it provides to Grafana in different ways; through an outside service, from a database or from memory.

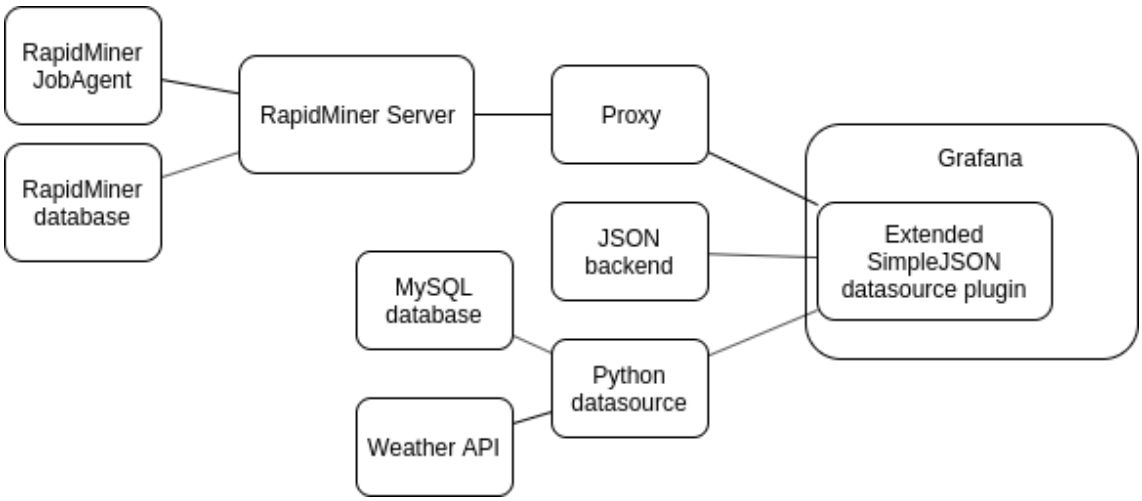

Figure 4.1: Architecture diagram

4.2 Components

In this section, each component's design is presented, focusing on their responsibilities.

4.2.1 Extended SimpleJSON datasource plugin

Grafana uses data source plugins to connect to different data storage backends. These components usually poll their backends, sending query requests to acquire the recorded information. Each data source plugin exposes a Grafana specific interface which allows Grafana to communicate with each data source plugin the same way.

The SimpleJSON datasource is made by the Grafana team and is available on GitHub (insert reference HERE).

It has two main purposes. One is to act as an example implementation to make writing custom data source plugins easier for the community. The other is to enable Grafana to read data from services that expose data in JSON format (which is widely popular thorough the industry).

For this thesis project, the latter role seems to be more important, as it possible to create a tool that receives data from different kind of formats and translates them into JSON. This way, we can send the transformed data to Grafana, which as a result would be able to display the collected data in one place that originally came from various sources.

Hence, instead of designing a data source plugin for Grafana from scratch, I chose to customize this already available component to dodge many caveats of developing an own plugin for a complex system.

The SimpleJSON data source plugin requires its backend to implement the following endpoints:

- */* - This endpoint is used for testing the connection between Grafana and the data backend. If everything is in order on the side of the backend, this endpoint should a HTTP 200 response.
- */search* - This endpoint is used for finding the available metric options. For example the names for different time-series.
- */query* - This endpoints is used to acquire the actual metrics data from the backend.

- */annotations* - This endpoint is used to return objects called annotations, additional information attached to metrics data. In the context of this project, we do not use this feature, however, it is needed for the data source plugin to run without errors.

In order to introduce additional interactivity capabilities, when integrating the RapidMiner Webservice, the SimpleJSON data source plugin needs to be extended. Since it is possible for a RapidMiner Webservice to accept parameters, which can filter the result, it would be useful to be able to acquire these available exposed parameters, so we can make more accurate metric queries with Grafana.

This means that the plugin has to be able to send another type of request to the backend, which in return would respond with the list of the accessible parameters:

- */parameters* - This endpoint is used for acquiring the available query parameters exposed by the backend.

During operation, this data source plugin will periodically poll its connected data sources for data, sending HTTP requests to the above described endpoints. The time-interval between the pools depends on the settings in Grafana.

4.2.2 Proxy

The main responsibility of this component is to translate between the RapidMiner Web service which the RapidMiner Server exposes and the SimpleJSON datasource plugin. The problem is that RapidMiner Server exposes its result in JSON, but is in another format that the SimpleJSON data source plugin accepts.

To accomplish that this proxy component could communicate with Grafana, it must implement the endpoints required by the SimpleJSON plugin. These were explained in the section 4.2.1 describing the plugin.

So the gateway component should be able to handle HTTP requests from SimpleJSON as well as forwarding them to RapidMiner after the translation. For this reason, it must be a constantly available service.

There are some RapidMiner web service specific considerations which need to be taken into account while designing the gateway component. We have to define the actions made by the proxy towards the RapidMiner Server in an abstract way, in order to properly implement the component.

4.2.2.1 Searching the targets

In order for Grafana to be able to make queries to request data, it needs to know where this data can come from, what is the address of the service that exposes the recorded information. For that, Grafana (and indirectly the SimpleJSON data source plugin) first asks for the available targets of the given data source backend. As it was mentioned in section 4.2.1, SimpleJSON uses the `/search` endpoint for acquiring this information.

In our case, the Proxy component connects to RapidMiner Server which exposes its data providing processes through web services. Thus, when accepting requests on the `/search` endpoint the gateway should return with the names of the available RapidMiner web services in a format that the data source plugin accepts. For that, it must send a request to the RapidMiner Server, asking for these names each time, so it is ensured that the information is always up-to-date. This is crucial, as the name of the RapidMiner web service determines the address where the gateway has to send the query requests later.

4.2.2.2 Acquiring the parameters

To enable the customization of the queries, the Proxy's `/parameters` endpoint should return the parameters exposed by a given web services. This information also falls in the category that should not be cached, as the possession of false knowledge can lead to failed queries that ends in no data displayed in Grafana.

After we have the name of the web service and its available parameters, we can finally request the actual data. For that, we use the `/query` endpoint. This is the place when the translation of different different data formats happens.

4.2.2.3 Querying the data

When Grafana needs to collect the actual data, it sends a request to the `/query` endpoint of the Gateway component. Grafana can handle data in two formats: time series, and table. This means that the Proxy component should be able to convert the acquired data from its backend into one of these formats.

4.2.3 JSON backend

This component is a example backend implementation for the SimpleJSON data source plugin. It serves as a base for creating other backends for the data source plugin. It also further expresses the general usability of the SimpleJSON plugin.

As this component can work with the SimpleJSON data source plugin out-of-the-box, I used it during the thesis project mainly for testing purposes. For example to check if the connection is still healthy between the components, or to inspect the specific data formats which are sent to or received from the SimpleJSON plugin.

4.2.4 Python data source

Although Grafana has multiple built-in plugins to communicate with databases, there exists some use-cases, when having a custom component between the data source and the visualization tool is feasible.

With an additional component in the middle, we have extra control over the data which travels from the data storage backend (in our case, MySQL) to the visualization platform. This means that we do not have to rely solely on the capabilities of the database, which can lead to simpler queries, smaller communication overhead with the database.

Having a custom middleware also makes it possible to implement the business logic in a separate component and only display business-relevant information with the visualization tool.

It also enables us to aggregate information from different backends and provide only one kind of interface towards visualization which can result in better maintainability. For example in our case, the Weather API component acts as a second backend.

Similarly as the Proxy component, the Python data source also needs to have those endpoints which are required by the SimpleJSON data source plugin, so this part is analogous to that in the Proxy. The effects of invoking them are slightly different than in case of the Proxy, because this Python source uses two components as backends, the 'external' API service and a separate database as described in section 3.2. So in order to get the requested data, the Python data source has to turn both of its backends. This means, that the Python data source has to posses more sophisticated communication capabilities. It needs to be able to handle requests from Grafana, send them back, make connections to the database and query its contents and to send requests to the 'external' API service and accept the results from it.

4.2.5 Weather API

As it was already described in section 3.2 describing the sample use-case, I implemented a simple service, that can return the all-time (calculated in a heuristic way, as it was already mentioned) maximum and minimum temperatures measured in New York for a given day of the year.

This component represents the previously mentioned external service. Its responsibility is to expose a service that can be utilized by other softwares in order to acquire additional business-relevant information.

It is worth to mention, that there are already several publicly available APIs that can be used to retrieve weather related data. For example, the API services of OpenWeatherMap or AccuWeather. However, these services are usually not free or only offer a free trial version which lasts just a couple a days long. This is why I decided to make an own API service that can be used during the whole time of preparing the thesis project.

— TODO — references for the weather apis

4.2.6 MySQL database

In section 3.2, it was stated, that one the backends used by the Python data source is a separate database. This database stores the data of the measured daily maximum and minimum temperatures in New York throughout a couple of years.

Chapter 5

Implementation

In this chapter, the previously described components are explained in further details, focusing on how they are implemented, what technologies were used and in addition, some extra fine tunes are presented which demonstrate the extendability of the open source Grafana.

The source code for the thesis project is available and can be viewed in my GitHub repository.

==TODO reference REPO==

5.1 Architecture

5.1.1 Docker

Throughout this project, I run the different components using Docker containers. For this reason, it seems appropriate to briefly introduce this technology.

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environ-

ment and ensure that it works uniformly despite differences for instance between development and staging.

=== TODO insert reference of Docker

5.1.2 Docker Compose

In order to be able to set up an architecture as the one described in section 4.1, I used Docker Compose.

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, we can use a YAML file to configure application's services. Then, with a single command, we can create and start all the services from the configuration.

The `docker-compose.yml` file for the project can be found in the GitHub repository of the thesis project. It defines all the necessary components with the appropriate configuration.

```
10    proxy:
11      build: ./proxy
12      ports:
13      - 5000:5000
14      restart: always
15      volumes:
16      - "./proxy/proxy.py:/code/proxy.py"
17
```

Listing 5.1: Extract of the `docker-compose.yml`

- === TODO insert reference of docker compose

5.2 Gateway

It was established in the design section 4.2.2, that the main responsibility of the Gateway component is to translate between the JSON data formats used by the RapidMiner Server and the Grafana SimpleJSON data source plugin.

For this task, I had two possibilities. One is to use tool which to define the transformation in a declarative way. For example, I could use XLST, which is able to convert a JSON format into another. The other option was to implement an own algorithm which handles the transformation of the data format. I chose the latter

alternative as the structure of the data was not exceptionally complex, thus it was quite simple to implement a function for this job . The other reason was that this way, I did not have to integrate another tool and call it every time to transform a simple JSON object, which in general reduced the number of possible sources for errors.

=== TODO reference XSLT tutorial

As it was described in section 4.2.2, the Gateway component must be able to handle requests from Grafana, as well as be able to forward these requests to the RapidMiner Server after the format translation.

For this task, I implemented the Gateway in Python using the `flask` and the `requests` packages and ran it in a Docker container. Flask is a lightweight web application framework library that enables quick and simple development. Requests can be thought of as the de facto standard for making HTTP requests in Python. It abstracts the complexities of making requests behind a simple API.

5.2.1 Endpoints

As it was explained in section 4.2.2, the Gateway component has to expose the endpoints that are required by the SimpleJSON data source plugin.

With the `requests` package, it is quite convenient to implement these endpoints, since we only have to declare a decorated function for each endpoint, as it is shown in the code snippet 5.2.

```
18  @app.route('/')
19  def check_connection():
20    response = requests.get(server_host)
21      if response.status_code == 200:
22        return 'Server connection OK'
23    return "Server connection error"
```

Listing 5.2: Test the connection to the server

5.2.1.1 Searching the targets

To be able to get the names of the available RapidMiner web services, the Gateway component should use the `/api/rest/service/list` endpoint of the RapidMiner Server. This endpoint returns the names and parameters of the web services in JSON format. The Gateway extracts the names of the services transforms this data into a format that SimpleJSON can understand.

The endpoint is implemented as displayed in the following 5.3 code snippet.

```
@app.route('/search', methods=['POST'])
def search():
  response = requests.get(server_host + '/api/rest/service/list',
                          auth=('admin', 'changeit'))
  webservices_json_list = json.loads(response.text)
  webservice_names = []
  for webservice in webservices_json_list:
    webservice_names.append(webservice['name'])
  return json.dumps(webservice_names)
```

Listing 5.3: Get the names of the web services

5.2.1.2 Acquiring the parameters

To get the available parameters of the exposed web services, the Gateway component uses the same endpoint of the RapidMiner Server (`/api/rest/service/list`), as for searching the targets.

The reason for that these two features are separated, even though they use the same endpoint, is that concerning the Gateway, the functionality of exposing the available targets is required by the SimpleJSON by default while querying the parameters is an extra feature. This means that SimpleJSON expects a specific data format, which only includes the names of the targets. Thus returning also the parameters with the targets would break the compatibility between the Gateway component and the SimpleJSON data source plugin.

Acquiring the parameters is implemented the following way, as the code snippet 5.4.

```
@app.route('/parameters', methods=['GET'])
def parameters():
  if request.args:
    args = request.args
    webserviceName = args.get('webserviceName')
    if webserviceName == None:
      raise ValueError('No value provided for "webserviceName"')
    # get the list of webservices and get the parameters of the one with the
  name provided in the query
    response = requests.get(server_host + '/api/rest/service/list',
    auth=('admin', 'changeit'))
    webservices_json_list = json.loads(response.text)
    webservice_params = []
    for webservice in webservices_json_list:
      if webservice['name'] == webserviceName:
        webservice_params = webservice['parameters']
        break
    return json.dumps(webservice_params)
  return 'Please provide a query parameter in the URL'

```

Listing 5.4: Get the parameters for a given web service

5.2.1.3 Querying the data

When Grafana sends a request to the Gateway component, it specifies the data format, in which it excepts to receive the results. This can be a time series format or a table format which can be seen in the code snippets below.

```
[
 {
    "target":"
  weather_retrieve_process?city=
  Vancouver",
    "datapoints":[
      [284.63, 1573882025382],
      [284.62904131,
  1573885625382],
      [284.626997923,
  1573889225382],
      [284.624954535,
  1573892890284],
      [284.622911147,
  1573896490284]
    ]
 }
]
```

Listing 5.5: Time series format

```
[
 {
   "columns":[
     {"text":"Month","type":"
  string"},
     {"text":"Maxs","type":"number
  "}
   ],
   "rows":[
     ['2012 October', 1],
     ['2012 November', 1]
   ],
   "type":"table"
 }
]
```

Listing 5.6: Table format

The time series response format contains the target name, which in this case is the name of the RapidMiner web service with an additional parameter attached and the data points. The first value in a data point is the value, the second is the timestamp in UNIX Epoch time in milliseconds.

The table format describes the name and the type of each column in the `columns` array. The actual data is in the `rows` array, every row is defined with a list of values in an order that is correspondent to that of the columns.

5.3 Python data source

As it was mentioned in section 3.2 and section 4.2.2, the Python data source communicates with a database and a separate API service in order to prepare the requested data for Grafana.

The Python data source is also connected to Grafana, so it needs to expose the endpoints that are needed by the SimpleJSON data source plugin (discussed in section 4.2.1), which are implemented similarly as in the Gateway.

The differing ability of the Python data source compared to the Gateway component, that it can connect to a MySQL database and an external API to get the requested data by Grafana. To be able to implement these features, I used the `requests` and

the `mysql.connector` packages along with the `flask` library, which is needed to be able HTTP requests (as in the case of the Gateway component).

Reading the data from the MySQL database is implemented as the following pseudo code shows (5.7).

```
def read_data_from_db():
  try:
    connection = mysql.connector.connect(
      <credentials>
    )
    sql_select_records_query = "SELECT * FROM newyork"
    cursor = connection.cursor()
    cursor.execute(sql_select_records_query)
    records = cursor.fetchall()
    data = []
    for row in records:
      record = {
        <parse data from a record>
      }
      data.append(record)
    return data
  except Error as error:
    print("Error reading data from MySQL ", error)
  finally:
    <close the connection to the database>

```

Listing 5.7: Reading data from MySQL

Getting the recorded information from the 'external' Weather API is much simpler thanks to the level of abstraction the `requests` package provides. (5.8)

```
def get_all_time_records():
  response = requests.get(weather_api_host + '/all')
  json_data = json.loads(response.text)
  return json_data

```

Listing 5.8: Reading data from the Weather API

After the Python data source possesses the data both from the MySQL and the Weather API, it aggregates the collected information (`create_table_data()` function) according to the sample business-logic already discussed in section 3.2 and then exposes it on its `/query` endpoint. (5.9)

```
@app.route('/query', methods=['POST'])
def query():
  table_data = create_table_data()
  return json.dumps(table_data)

```

Listing 5.9: Exposing the prepared data

5.4 Additional fine tunes

5.4.1 Customized Grafana GUI

5.4.2 Dynamic bar chart in Grafana

Chapter 6

Evaluation

6.1 Pros

6.2 Cons

Chapter 7

Future work

- Integrate proxy and Grafana RapidMiner datasource into one

Chapter 8

Related works

interactive-piechart-panel (github/eastcirclek) see notebook

Chapter 9

Summary

- data format conversion is important for integrating complex systems, thus these kind of gateways are crucial

Acknowledgements

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Bibliography

[1] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar. Diplomaterv portál (2011. február 26.). `http://diplomaterv.vik.bme.hu/`.

[2] James C. Candy. Decimation for sigma delta modulation. 34(1):72–76, 01 1986. DOI: `10.1109/TCOM.1986.1096432`.

[3] Gábor Jeney. Hogyan néz ki egy igényes dokumentum? Néhány szóban az alapvető tipográfiai szabályokról, 2014. `http://www.mcl.hu/~jeneyg/kinezet.pdf`.

[4] Peter Kiss. Adaptive digital compensation of analog circuit imperfections for cascaded delta-sigma analog-to-digital converters, 04 2000.

[5] Wai L. Lee and Charles G. Sodini. A topology for higher order interpolative coders. In *Proc. of the IEEE International Symposium on Circuits and Systems*, pages 459–462, 05 4–7 1987.

[6] Alexey Mkrtychev. Models for the logic of proofs. In Sergei Adian and Anil Nerode, editors, *Logical Foundations of Computer Science*, volume 1234 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63045-6. DOI: `10.1007/3-540-63045-7_27`. URL `http://dx.doi.org/10.1007/3-540-63045-7_27`.

[7] Richard Schreier. *The Delta-Sigma Toolbox v5.2*. Oregon State University, 01 2000. `http://www.mathworks.com/matlabcentral/fileexchange/`.

[8] Ferenc Wettl, Gyula Mayer, and Péter Szabó. *LaTeX kézikönyv*. Panem Könyvkiadó, 2004.