Do Bao Khanh

DATA VISUALIZATION WITH INTEGRATION OF GRAFANA AND QUANTA PLATFORM

Technology and Communication
2021

FOREWORD

I am always grateful for spending 4 years studying at Vaasan Ammattikorkeakoulu, University of Applied Sciences. I believe what I learnt here will go with me forever.

I would like to show my appreciation to Dr. Ghodrat Moghadampour, my supervisor, for supporting me with my thesis.

I would give my gratitude to Mr. Matti Rita-Kasari, Vilppu Vuorinen, and Luukas Kuusrainen for their massive support. Without them, I could not make this happen.

Furthermore, I would like to send my love and thanks to my family and my friends who always stay by my side and support me.

Vaasa 25/03/2021

Do Bao Khanh

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Do Bao Khanh
Title	Data Visualization with Integration of Grafana and Quanta Platform
Year	2021
Language	English
Pages	50
Name of Supervisor	Ghodrat Moghadampour

The objective of this thesis was to develop a data source plugin which was integrated with the Quanta platform to perform data visualization on a web application. Grafana is a multi-platform analytics, which is used to build interactive web applications which allow data visualization. Quanta is an open-source data analytics platform focusing on the seamless integration between algorithm development and business systems.

The Grafana data source plugin was developed by using TypeScript, ReactJs, and Software Development Kits provided by Grafana. In the plugin, data is fetched by using APIs provided by Quanta. Quanta performs the data analytics and visualizes the data on the Grafana dashboard.

The project resulted in a data source plugin that can use the Quanta concepts, APIs to query data from Quanta, and analyze data. With the developed data source plugin, users can have better data visualization, query more complex data queries, compare different data sources together, and utilize features provided by Grafana.

The data source plugin has been deployed and is currently in the customer test phase.

Keyword	Grafana, Quanta, data analyst, data source plugin, data visualization

CONTENTS

ABSTRACT

1 INTRODUCTION 1

2 TECHNOLOGIES BACKGROUND 3

2.1 JavaScript 3

2.2 ReactJS 3

2.3 TypeScript 4

2.4 TimescaleDB 5

2.5 RESTful API 6

2.6 Docker 7

2.7 Quanta 8

2.8 Grafana 10

3 APPLICATION DESCRIPTION 13

3.1 Grafana Data Source Plugin Overview Structure 13

3.2 Requirements Specification 14

3.3 Use-Case Diagram 15

3.3.1 Data Source Configuration Use-case Diagram 15

3.3.2 Query Configuration Use-Case Diagram 16

3.3.3 Data Visualization Use-Case Diagram 17

3.3.4 Variable Query Configuration Use-case Diagram 18

3.4 Functions Description 19

3.4.1 Data Source Configuration Sequence Diagram 19

3.4.2 Query Configuration Sequence Diagram 20

3.4.3 Data Visualization Sequence Diagram 21

3.4.4 Variable Query Configuration Sequence Diagram 21

4 GUI DESIGN 23

4.1 Config Editor 23

4.2 Query Editor 24

4.3 Variable Query Editor 25

5 IMPLEMENTATION 27

5.1 Implementation overview structure 27
5.2 Setting up Environment 29
5.3 Anatomy of Data Source Plugin 31
5.4 Config Editor 33
5.5 Query Editor 35
5.6 Data Source 37
5.7 Variable Query Editor 42
6 TESTING THE DATA SOURCE PLUGIN 43
7 CONCLUSION 48
7.1 Future Work 48
REFERENCES 49

LIST OF FIGURES AND TABLES

Figure 1. Quanta workflows p. 9

Figure 2. Data Source Configuration use-case diagram p. 15

Figure 3. Query Configuration use-case diagram p. 16

Figure 4. Data Visualization use-case diagram p. 17

Figure 5. Variable Query Configuration use-case diagram p. 18

Figure 6. Data Source Configuration Sequence Diagram p. 19

Figure 7. Query Configuration sequence diagram p. 20

Figure 8. Data Visualization sequence diagram p. 21

Figure 9. Variable Configuration sequence diagram p. 22

Figure 10. Config Editor GUI p. 23

Figure 11. Query Editor GUI p. 24

Figure 12. Variable Query Editor GUI p. 25

Figure 13. Grafana data source plugin workflows p. 28

Figure 14. Data source configuration example p. 43

Figure 15. Series data query example p. 44

Figure 16. Series data visualization example p. 44

Figure 17. Result output data queries example p. 45

Figure 18. Series data and the forecasted value in the same panel p. 45

Figure 19. Variables in the panel example p. 46

Figure 20. Variable syntax example p. 46

Figure 21. Data visualization with the variables p. 47
Figure 22. Dashboard with multiple panels p. 47
Table 1. Data source plugin requirement p. 14

LIST OF CODE SNIPPETS

Code Snippet 1. TypeScript type-checking example p. 5

Code Snippet 2. TimescaleDB queries data example p. 6

Code Snippet 3. Docker-compose.yml p. 29

Code Snippet 4. Command to run Docker-compose p. 30

Code Snippet 5. Create the project folder with grafana-toolkit p. 31

Code Snippet 6. plugin.json p. 32

Code Snippet 7. module.ts p. 32

Code Snippet 8. Config Editor UI component p. 35

Code Snippet 9. Query Editor component p. 36

Code Snippet 10. query() method p. 39

Code Snippet 11. testDatasource() method p. 40

Code Snippet 12. metricFindQuery() method p. 42

Code Snippet 13. Variable Query Editor component p. 42

LIST OF ABBREVIATIONS

JDK - Java Development Kit

REST – Representational state transfer

API – Application Program Interface

HTML – Hypertext Markup Language

CSS – Cascading Style Sheets

DOM – Document Object Model

UI – User Interface

SQL – Structured Query Language

HTTP – Hypertext Transfer Protocol

SDK – Software Development Kit

CLI – Command-line Interface

JDBC – Java Database Connectivity

CSV – Comma-separated values

JSON – JavaScript Object Notation

SSL – Secure Sockets Layer

IP – Internet Protocol

GUI – Graphical User Interface

ISV - Independent Software Vendors

1 INTRODUCTION

With the development of technology, data is growing exponentially and has kept an important role in any business decision. Data has been used to answer questions and solve problems. Through data analysis, we can predict the future more accurately than ever before. The world is witnessing a transformation of the traditional business to the innovative business where data can support the decision-making.

Data visualization has been challenging many companies in recent years. Quanta, which is an open-source data analytics platform developed by Jubic Oy with a focus on algorithm development and seamless integration between business systems has encountered the problem of how to visualize the data in the most efficient way /1/. Grafana, which is an open-source platform, comes to the rescue for data visualization. Grafana has become the most popular technology used to compose observability dashboards. Thousands of companies have been using Grafana in their operations such as PayPal, eBay, and Intel. Grafana answers the question of how to visualize data excellently. It is completely open source and backed by a vibrant community where hundreds of dashboards and plugins are already available in the official library. Grafana brings everyone together and shares data and dashboards among teams. /2/

This thesis work was carried out to develop a Grafana data source plugin that can fetch the data from Quanta platform and visualize them on Grafana, so that Quanta can focus on their algorithms, while Grafana handles the visualization, query data, and analyse the data. With the various features of Grafana, the data source plugin can take advantage of those features. The application is very extensive when Grafana allows developers to build their own data source plugins.

This report is divided into the following parts:

- The technology stack used in this project.

- The application description
- The GUI designs
- The implementation of the Grafana data source plugin
- Production, and testing.

2 TECHNOLOGIES BACKGROUND

This chapter briefly explains the technologies stack used in this project. Quanta platform backend used Java with JDK 11 to develop the backend, and TimescaleDB as the database. Grafana used JavaScript, ReactJs, Typescript, and the development kit provided by it to build data source plugins. Grafana communicated with the Quanta backend through the RESTful API, provided by Quanta. Both Grafana and Quanta were built with Docker and published at Docker-hub

2.1 JavaScript

JavaScript is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, alongside HTML and CSS, JavaScript turns into one of the most popular web programming languages. The widespread majority of websites use JavaScript for the client side, and all the major web browsers have a dedicated engine to execute it. /4/

More than 97% of websites use JavaScript on the client side to define web pages, usually using third-party libraries. All popular web browsers have a dedicated JavaScript engine that allows code to run on the user's device. /4/

2.2 ReactJS

React (or React.js, ReactJS) is an open-source, JavaScript library for building reusable user interfaces or UI components. It is maintained by Facebook and backed up by a passionate community. React focuses on the state management and rendering that state to DOM. React can be used to develop single-page applications or mobile applications. /5/

React provides the easier way to create interactive UIs. React will efficiently update and render just the right components when the state changes. Declarative views make the code more predictable and easier to debug. /6/

Using ReactJS, developers can decompose complex UI structures into independent components. This idea helps developers not have to worry about general web applications, but only need to focus on decomposing complex UI structures into simpler components. This makes everything more intuitive and easier to understand than before.

2.3 TypeScript

TypeScript is a programming language developed and maintained by Microsoft. It is built on top of JavaScript with a strict syntactical and adds optional static typing to the language. TypeScript is designed for the development of large applications and trans compiles to JavaScript. Existing JavaScript programs are also valid TypeScript because TypeScript is a superset of JavaScript /7/.

Types provide a way to describe the shape of the object, providing better documentation, and allowing TypeScript to verify that the code is working correctly. Writing types can be optional in TypeScript because type inference provides many options without writing additional code. TypeScript code may encounter type checking errors, but they will not prevent the developers from running the generated JavaScript /8/.

TypeScript verifies JavaScript code ahead of time. The type-checking is important when building large-scale applications. Imagining when the application gets larger over the time, when we have thousands of variables, and more complex variable types, the maintenance can be painful for the developers. TypeScript comes to the rescue, with TypeScript we can build more valid code, and it is easier to maintain the code. The example below shows how TypeScript verifies the code before execution.

```
const user = {
  firstName: "Angela",
  lastName: "Davis",
```

```
  role: "Professor"
}
console.log(user.name)
Property 'name' does not exist on type '{ firstName: string; last-
Name: string; role: string; }'.
```

Code Snippet 1. TypeScript type-checking example /8/.

As we could see from the code snippet above, the variable *user* does not have the property *name*, so TypeScript will do the type-checking and return the error *Property 'name' does not exist on type '{ firstName: string; lastName: string; role: string; }' .* With the type-checking we could check the type before executing the JavaScript and reduce the mistakes when coding. To take advantage of this, most of the functions of this project were developed by using TypeScript.

2.4 TimescaleDB

TimescaleDB is a database for SQL scaling time-series data. It is based on PostgreSQL and is packaged as a PostgreSQL extension. It provides automatic time and space partitioning and completely supports SQL. TimescaleDB exposes what look like regular tables, but in fact it is just an abstraction (or a virtual view) of many separate tables that hold actual data. This single-table view is composed of many chunks, which are created by dividing the data in hypertables into either one or two dimensions: by a time-interval, and by an (optional) "partition key", such as device id, location, or user id. /9/

The following reasons answer why should we use the TimeScaleDB provided by TimeScale Official Website /14/:

- Completely SQL, impeccable reliability, and a huge ecosystem
- The speed of receiving request is 10 to 100 times faster than PostgreSQL, InfluxDB, and MongoDB
- Each node writes millions of data points per second. Horizontally scale to petabytes

- Simplifies stack, asks more complex questions, and builds more powerful applications.

Data can be queried from a hypertable by using a standard SQL command. The following example shows how we could use the standard SQL command to query the data.

```
-- Return the most recent 100 entries in the table 'conditions' ordered newest to oldest

SELECT * FROM conditions ORDER BY time DESC LIMIT 100;


-- Return the number of data entries written in past 12 hours

SELECT COUNT(*) FROM conditions

  WHERE time > NOW() - INTERVAL '12 hours';
```

Code Snippet 2. TimescaleDB queries data example /13/.

As seen in the code snippet above, the most recent 100 entries in the table **"condition"** is queried with the **SELECT, WHERE ORDER BY, DESC, LIMIT** keywords, which are the standard SQL keywords. We can use the aggregation as we used to in SQL.

In this project, TimescaleDB is used as the default database for the Quanta back end.

2.5 RESTful API

Representational State Transfer (REST) is a software architecture that uses a subset of HTTP. It is widely used to build interactive applications that use Web services. Web services that follow these guidelines are called RESTful. Such a Web service should provide its web resources in a textual representation and allow them to be read and modified with a stateless protocol and a set of predefined operations. RESTful systems are designed to achieve fast performance, reliability,

and grow potential through reusable components that can be managed and updated without affecting the entire system, even while it is running. /10/

The RESTful API divides a transaction into several small modules. Each module contains the main part of the transaction. This modularity provides developers with a lot of flexibility, but for developers, it can be difficult to build their REST API from scratch. The state of a resource at a specific time stamp is called a resource representation. RESTful API uses existing HTTP methods defined by the RFC 2016 protocol, such as /16/:

- GET: to retrieve a resource.
- PUT: to change the state of or update a resource.
- POST: to create that resource
- DELETE: to remove it.

Data formats the REST API supports includes:

- application/json
- application/xml
- application/x-wbe+xml
- application/x-www-form-urlencoded
- multipart/form-data

In this project, Grafana fetches the data from Quanta platform via RESTful API.

2.6 Docker

Docker is an open platform for developing, shipping, and running applications, which allows you to separate the applications from the infrastructure so that the applications can be delivered quickly. With Docker, we can package and run applications in a loosely isolated environment called a container. The isolation and security allow running many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so we do

not need to rely on what is currently installed on the host. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, users can significantly reduce the delay between writing code and running it in production. /11/

Docker provides tooling and a platform to manage the lifecycle of the containers /11/:

- Developing an application and its supporting components using containers
- The container becomes the unit for distributing and testing applications
- Deploy the application into a production environment, as a container or an orchestrated service.

Docker Hub is the world's largest container image repository, which contains a variety of content sources, including developers from the container community, open-source projects, and independent software vendors (ISV) that create and distribute their containerized code. Users can access a free public repository for storing and sharing images or can choose to subscribe to a private repository /12/.

Docker has been used to ship and run the applications in both Quanta and Grafana platforms.

2.7 Quanta

Quanta is an open-source data analytics platform developed by Jubic Oy, focusing on the seamless integration between algorithm development and business system/2/.

Any business data, or device data can be ingested by the Quanta platform, and Quanta has its own Quanta SDK which contains classical algorithms and machine learning to process and analyze the data. Quanta provides a way to filter, pre-pro-

cessing and visualize the input data. It also supports business intelligence and decision making by forecasting and predicting the future and moreover, supports anomaly detection which could help the customers avoid unexpected behavior.

Quanta needs to set up data connections to analyze the data with its own algorithms. The data connections could be JDBC databases, CSV files or JSON files which have clean data sets. Quanta ingests the data into a proper format defined by Quanta and adds raw data to the TimescaleDB. To use machine learning and algorithms to analyze it, Quanta provides an interface to register the machine learning programs called Worker. It could be a standalone program, writing any programming language and providing the APIs to which Quanta can connect to that Worker. To use a Worker, Quanta needs to authorize one to guarantee its validity. We can have multiple Workers to implement different tasks with different specifications. After authorizing Workers, they are available and ready to be used. The figure below shows the workflows of Quanta platform.

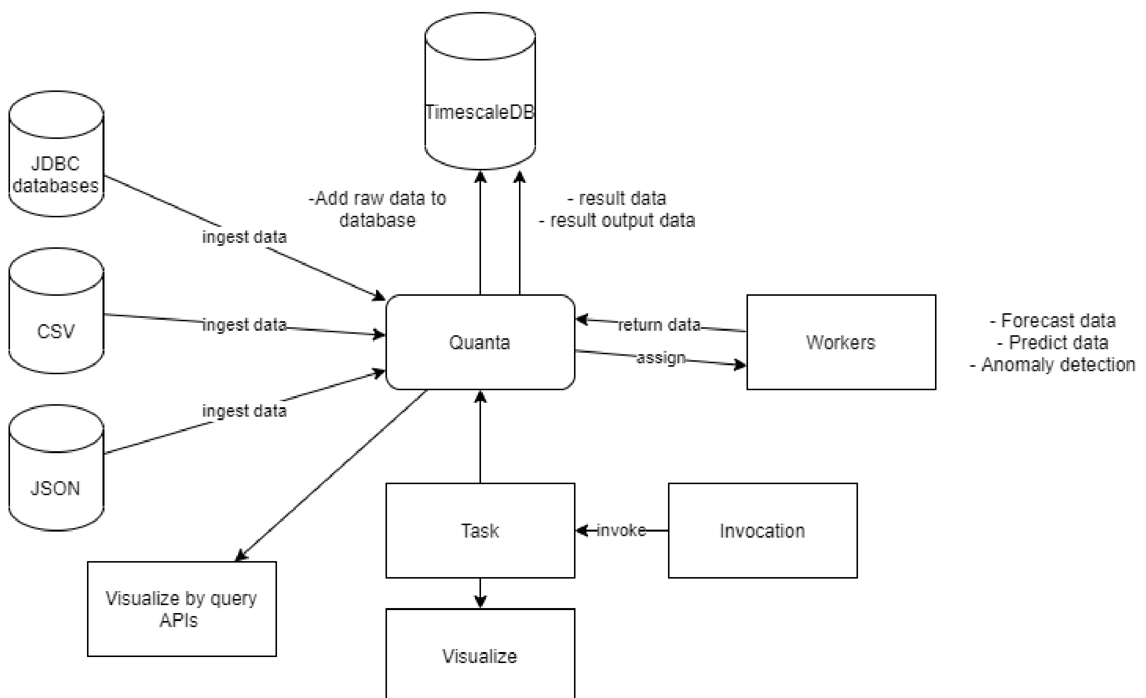

Figure 1. Quanta workflows

After having the data connections and the Workers, we can start defining a task which assigns the proper Worker to proper data connection depending on the purposes. In Task, we need to specify the data connection and the Worker. After selecting the Worker, its output columns will be shown, and we need to specify its input columns to create a task. In Task, we can also specify the parameters which are defined by the Worker. To pass the customized parameters, workers will get the parameters and do their work. Task is mutable meaning we can edit and change the information in the task. Accordingly, we need Invocation which takes a snapshot of the task before passing to the Worker. Invocation is immutable which has all the information of the task before passing to the worker.

After invoking the task, the worker will run its algorithm to analyze the data, pass back the result and result output data to Quanta, and store data in the Quanta database. The result data is a snapshot of data before sending to the Worker, and the result output is the result after Worker running algorithms to analyze the data. Users can visualize the data by a specific task by the query API, which queries directly to the database.

Quanta provides a way to integrate with other platforms by providing a client-token which is used as the authentication and authorization to a specific task, or to the query API. Users can generate tokens and attach to them to a specific task, or the tokens will be the global scope by default. Users need to keep those tokens in a safe place because once we have the token, we can access the Quanta database. Query API is the way Grafana communicates with another platform.

2.8 Grafana

Grafana is an open-source platform that provides features to query, visualize, and understand the metrics. Grafana has answered the questions of how to visualize data, which has been challenging many developers in recent years. Grafana brings all the data together and shares them among teams. Through various plugins, users could get what suits them most, or even build plugins to suit their needs.

Grafana is backed by a vibrant community with a passion to make Grafana better /2/.

Grafana offers the following features /15/:

- Data Visualization: Fast and flexible client-side graphs with many options. Metrics and logs will be visualized in the Panel
- Dynamic Dashboard: Reusable dynamic dashboards with multiple panels. The template variables displayed at the top of the dashboard.
- Explore Metrics: Compare different data sources, and queries by splitting views. Explore the data through ad-hoc queries and dynamic drilldown
- Explore Logs: Quickly search through all the logs or stream them live by switching from metrics to logs with preserved label filters.
- Alerting: Keep evaluating and sending notification systems such as Slack, or PagerDuty
- Mixed Data Sources: Different data sources can be mixed in the same graph.
- Annotations: Annotate event graphs from various data sources. Event tracking shows complete metadata and event tags
- Ad-hoc Filters: Creating new key-value on the fly, which will automatically be applied to all queries that use that data source by using Ad-hoc filters.

In addition to providing a large number of visualizations and data sources immediately after installing Grafana, we can use the plugins that extend the functionality of Grafana. We can install one of the plugins built by the Grafana community or create a plugin. Grafana supports three types of plugins: panels, data sources and apps /17/.

- **Data source plugins**: External data sources interact with the Data source plugin and return data in a format that Grafana understands. We can immediately start using data in any existing dashboard by adding a data source /18/.

- **Apps**: Data sources and panels are combined by using Apps plugin to provide a consistent user experience. Use Apps plugin when creating a custom monitoring experience /19/.
- **Panels**: The Dashboard can have new visualization types by adding Panel plugin. Use Panel plugins to visualize data returned by data source queries, navigate between dashboards, or control external systems /20/.

3 APPLICATION DESCRIPTION

This chapter briefly explains the analysis phase. The overview structure of Grafana data source plugin platforms will be explained in this chapter. Moreover, the modules, and requirements of the application will be described.

3.1 Grafana Data Source Plugin Overview Structure

Grafana plugins come in different shapes and sizes, however, there are some shared properties among them. Every plugin requires at least two files:

- **plugin.json:** contains information about the plugin
- **module.ts:** exposes the implementation of the plugin

To make Grafana detect the data source plugin, and make it work, the following components need to be implemented:

- **Configuration Editor**: the configuration where users need to set up the host, port, SSL flag, and the client token which is generated by Quanta.
- **Query Editor**: constructs queries that return to Data Source to fetch data from Quanta.
- **Data Source**: queries the data fetch them from Quanta and visualizes them in the Panel.
- **Variable Query Editor**: defines the variable query, or variable values that return to Data Source to apply variables to all the queries, or specific queries.

3.2 Requirements Specification

The requirements specification shows which functions must be implemented, and in which order. All the required functions are specified in this chapter.

Table 1. Data source plugin requirement

References	Description	Priority (1. Must have, 2. Should have, 3. Nice to have)
F1	Implement configuration editor, config the host, port, SSL flag, and client tokens	1
F2	Implement health check for data source, and the data source configuration	1
F3	Implement FROM CLAUSE, and SELECT CLAUSE in Query Editor	1
F4	Map the data from Quanta to Grafana data frames concepts	1
F5	Implement WHERE in Query Editor	2
F6	Implement GROUP BY CLAUSE in Query Editor	2
F7	Implement variables support	3

3.3 Use-Case Diagram

The use-case diagram below is used to describe the relationship between the actor and use-cases. The figure below shows the use-case diagram of the data source plugin.

3.3.1 Data Source Configuration Use-case Diagram

The figure below describes how the user can set up the connection to the Quanta.

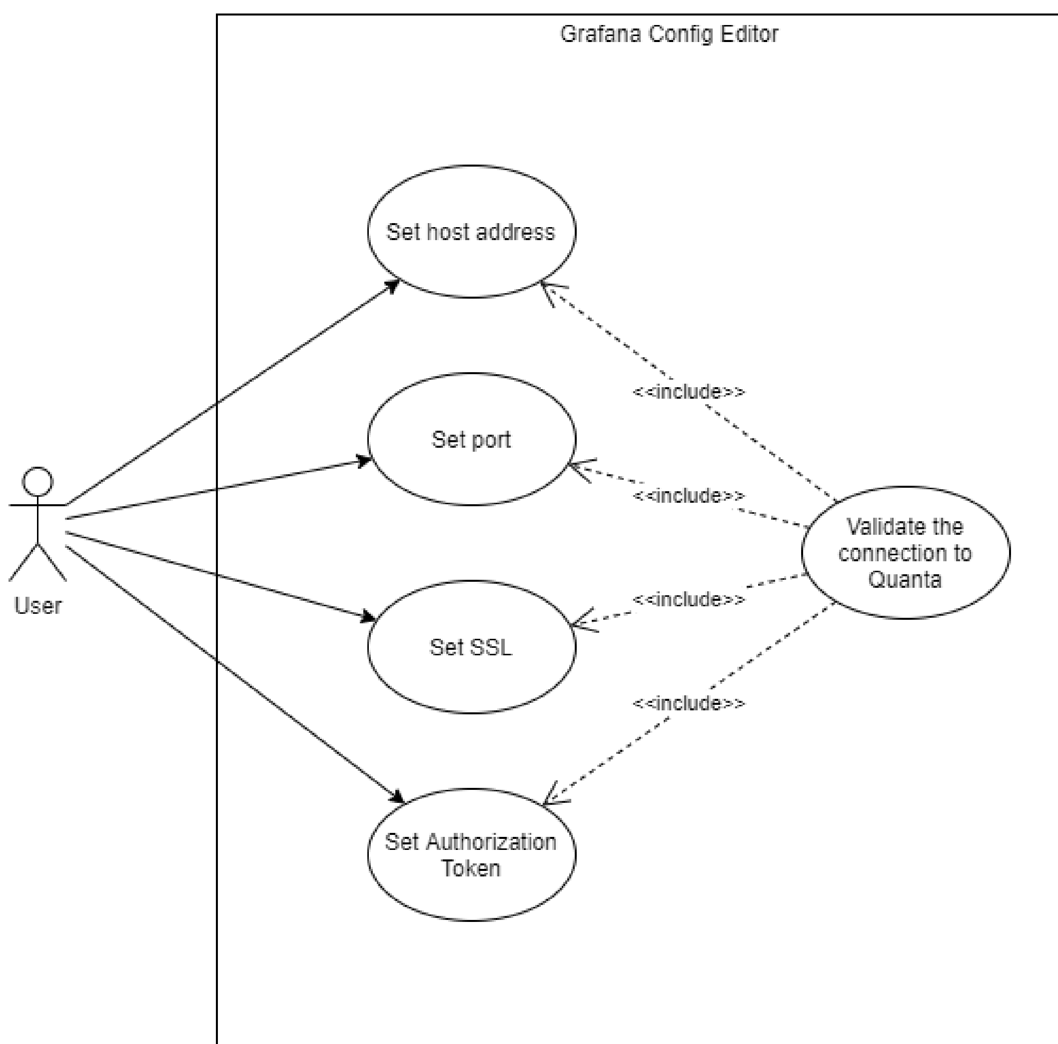

Figure 2. Data Source Configuration use-case diagram

The actor is the user who wants to set up the connection between Grafana and Quanta. The user needs to configure the following properties before validating the connection:

- Host address: the host address of Quanta endpoint
- SSL: enable/disable SSL
- Port: port of the Quanta endpoint
- Authorization: Authorization token

After configuring all the required fields, users can validate the connection.

3.3.2 Query Configuration Use-Case Diagram

The figure below shows the use-case diagram of how to construct the query.

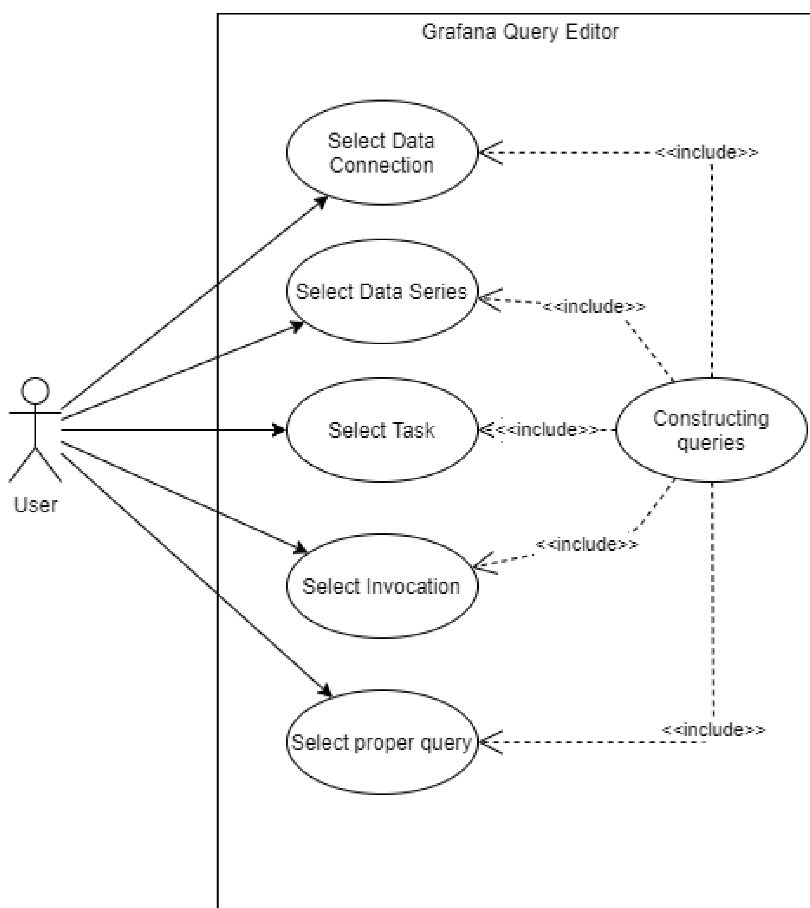

Figure 3. Query Configuration use-case diagram

Before selecting the query, the user needs to specify which Data Connection, Data Series, Task, and Invocation will be queried. After having the necessary properties, the query editor can start suggesting the proper query, so that the user can just select the available query in the dropdown list.

3.3.3 Data Visualization Use-Case Diagram

The figure below describes the use-case diagram of visualizing the data.

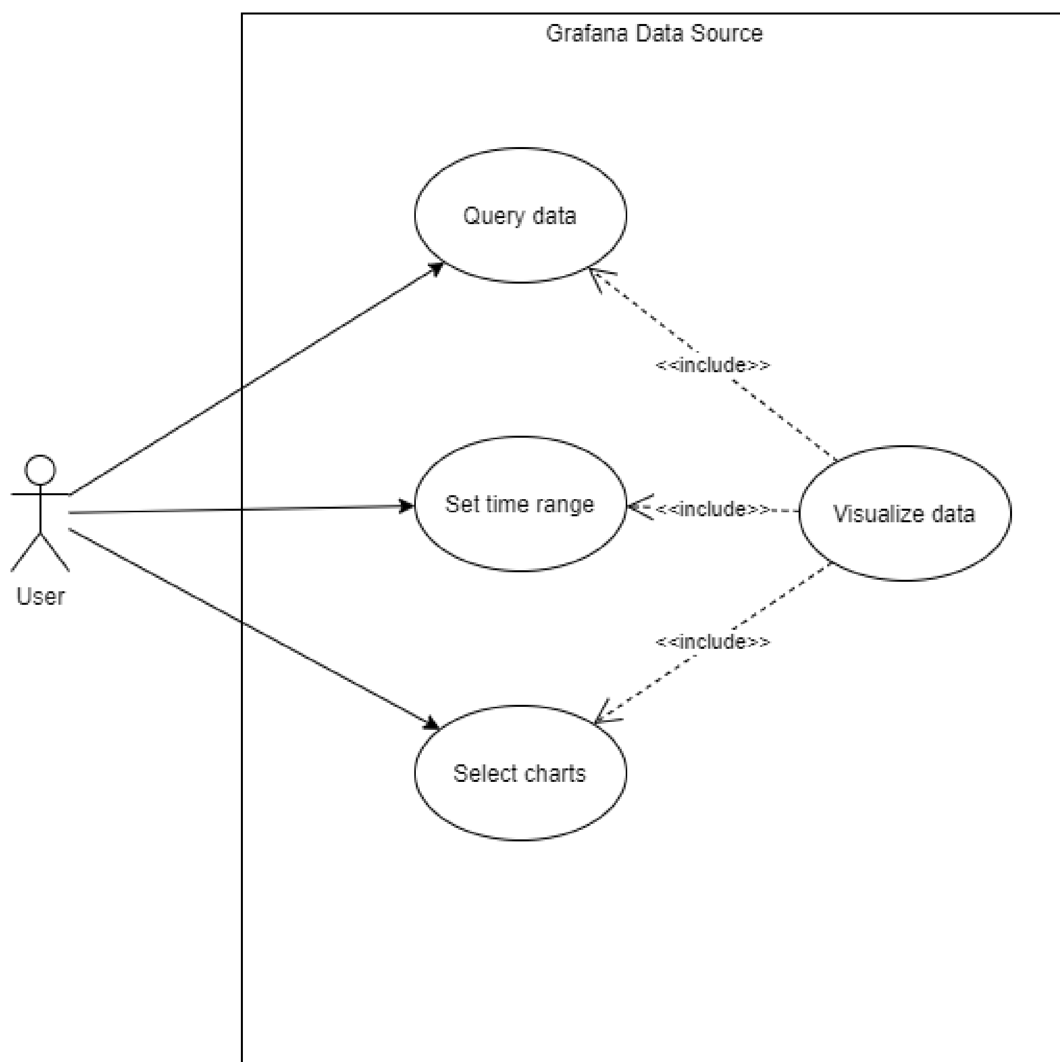

Figure 4. Data Visualization use-case diagram

After constructing the query in the Query Editor, Data Source will start querying the data, the user can set the time range, and select the chart. The data will be

visualized on the Panel, and users can select various charts provided by Grafana such as bar chart, pie chart, or line chart.

3.3.4 Variable Query Configuration Use-case Diagram

The figure below describes the use-case diagram of variable query configuration

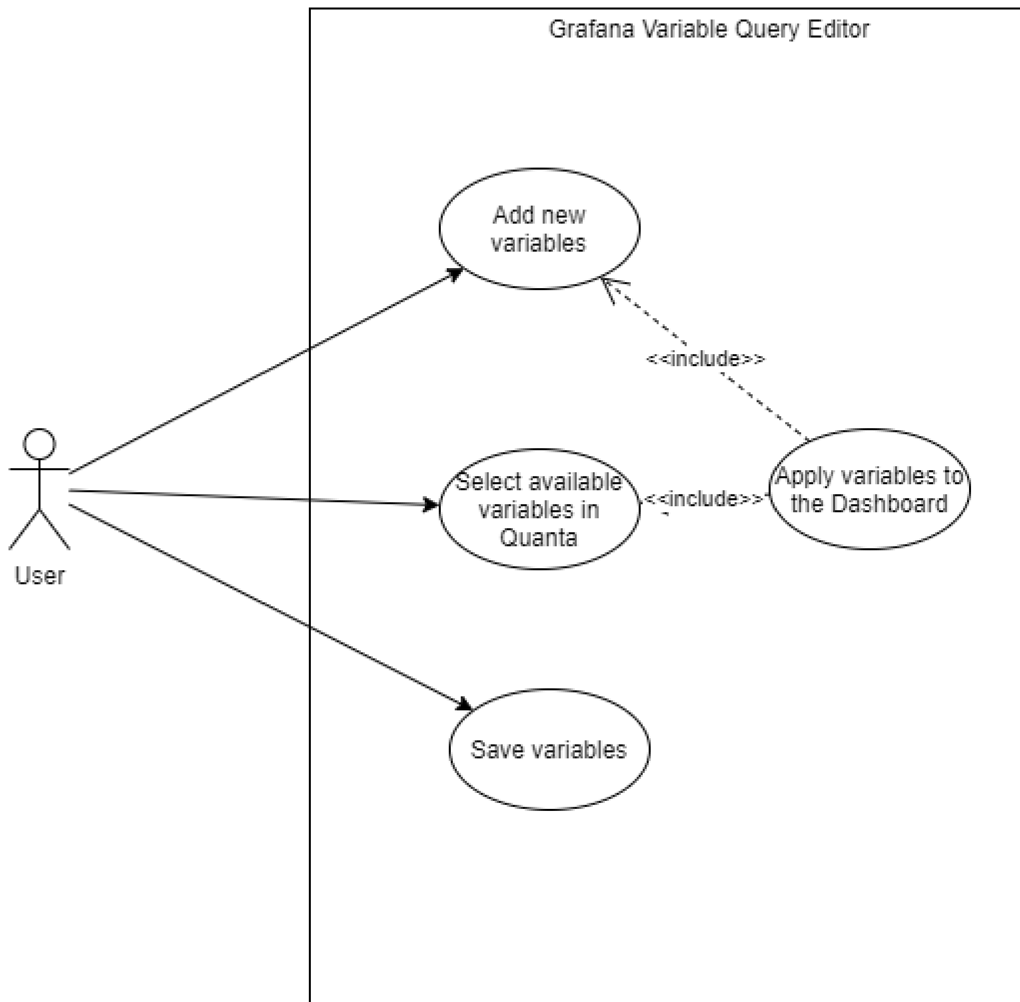

Figure 5. Variable Query Configuration use-case diagram

Users can add new variables manually with the form provided by Grafana. If users are not certain which variables to add, Variable Query Editor will suggest the available queries to fetch the available variables in Quanta. Users can select the proper query from the suggestion. After having the variables, users can save variables for later usage.

3.4 Functions Description

To describe functions in detail we use a sequence diagram to show how each function has been designed.

3.4.1 Data Source Configuration Sequence Diagram

The user needs to set up the connection between Quanta and Grafana. The figure below shows the process of setting up the connection.

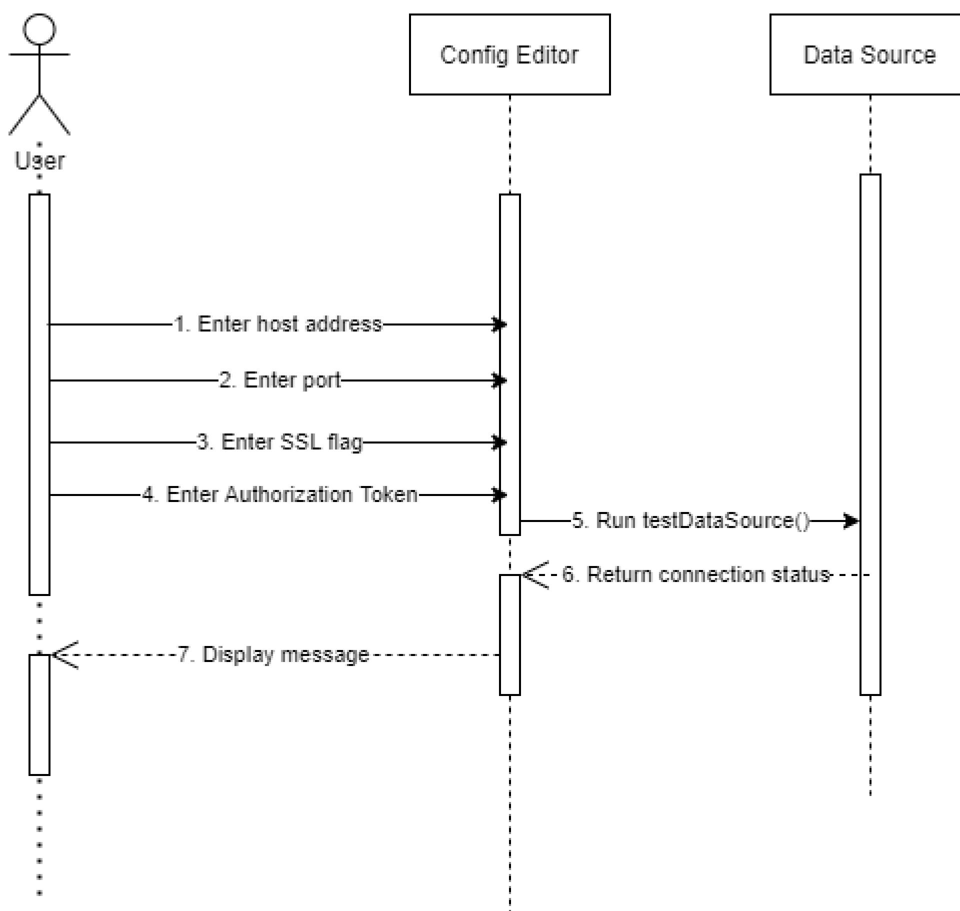

Figure 6. Data Source Configuration Sequence Diagram

The user needs to enter the host address, port, SSL flag, and Authorization Token to run the health check for the data source. After the user enters all the required fields, Data Source will run the *testDataSource()* to check the connection, and return the status of the connection to the user.

3.4.2 Query Configuration Sequence Diagram

After configuring the data source connection between Grafana and Quanta (Figure 6), the figure below shows the process of constructing queries.

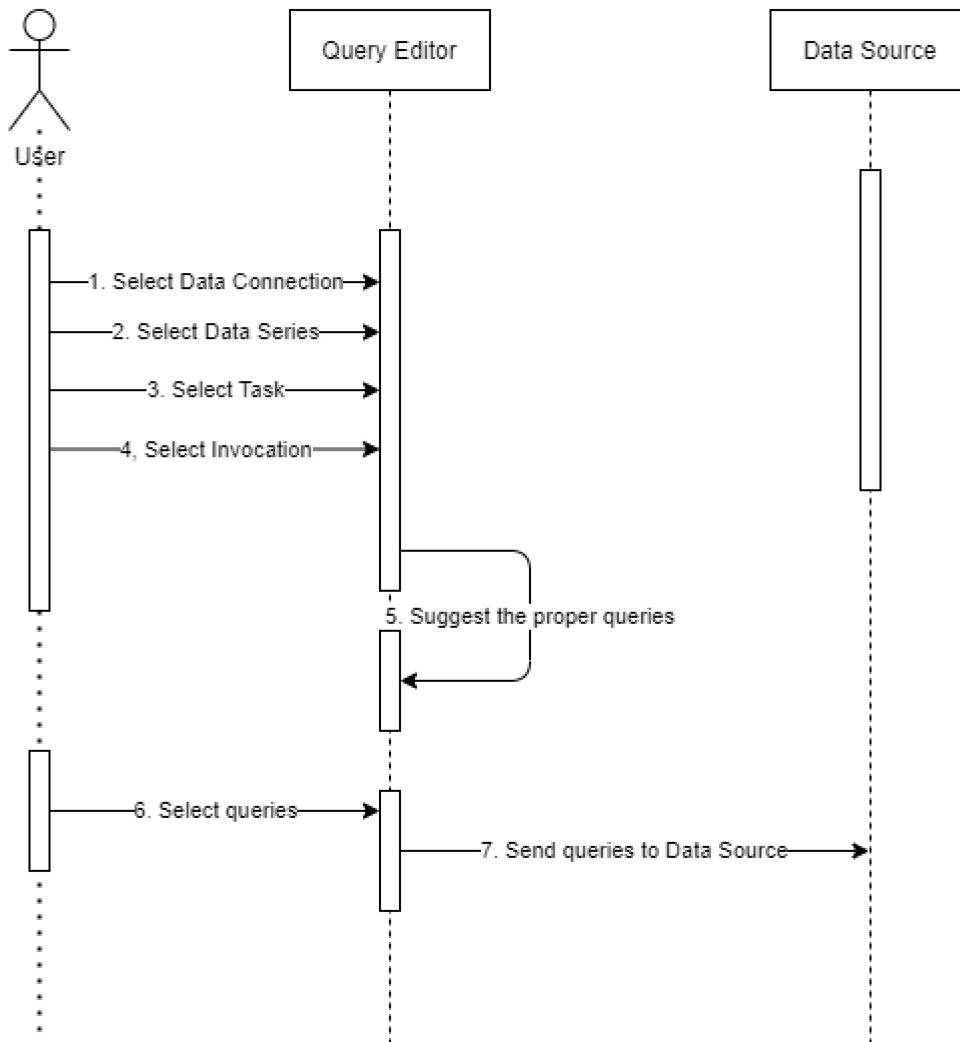

Figure 7. Query Configuration sequence diagram

The user needs to select the data connection, data series, task, and invocation, to help the Query Editor to understand what the user is querying. After having enough information to suggest the proper queries, the user can start selecting queries, and all the queries will be sent to Data Source to fetch the data.

3.4.3 Data Visualization Sequence Diagram

After getting queries from Query Editor (Figure 7), Data Source will fetch the data and visualize it on the Grafana Panel. The figure below shows the function description of visualization of data. After fetching data from Quanta, data will be visualized with the default time range of Grafana, and the default chart is the line chart. Users can set the time and select another chart available in Grafana.

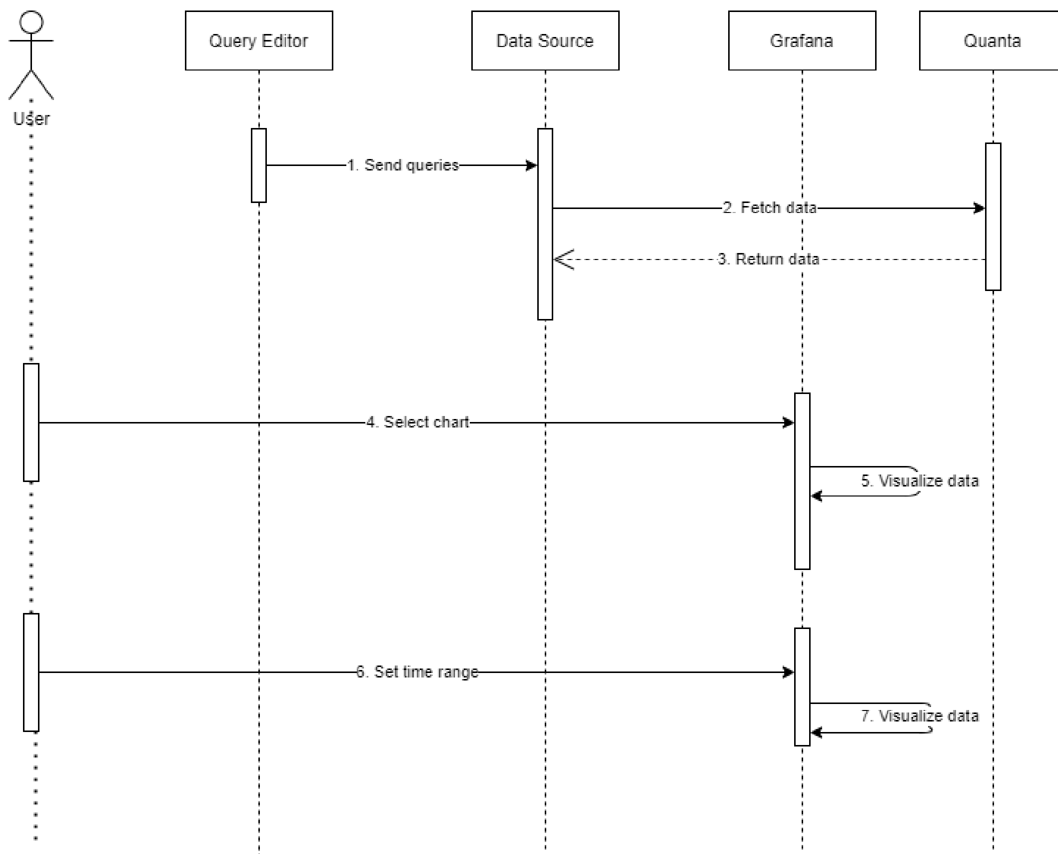

Figure 8. Data Visualization sequence diagram

3.4.4 Variable Query Configuration Sequence Diagram

The figure below describes the function descriptions of creating variables.

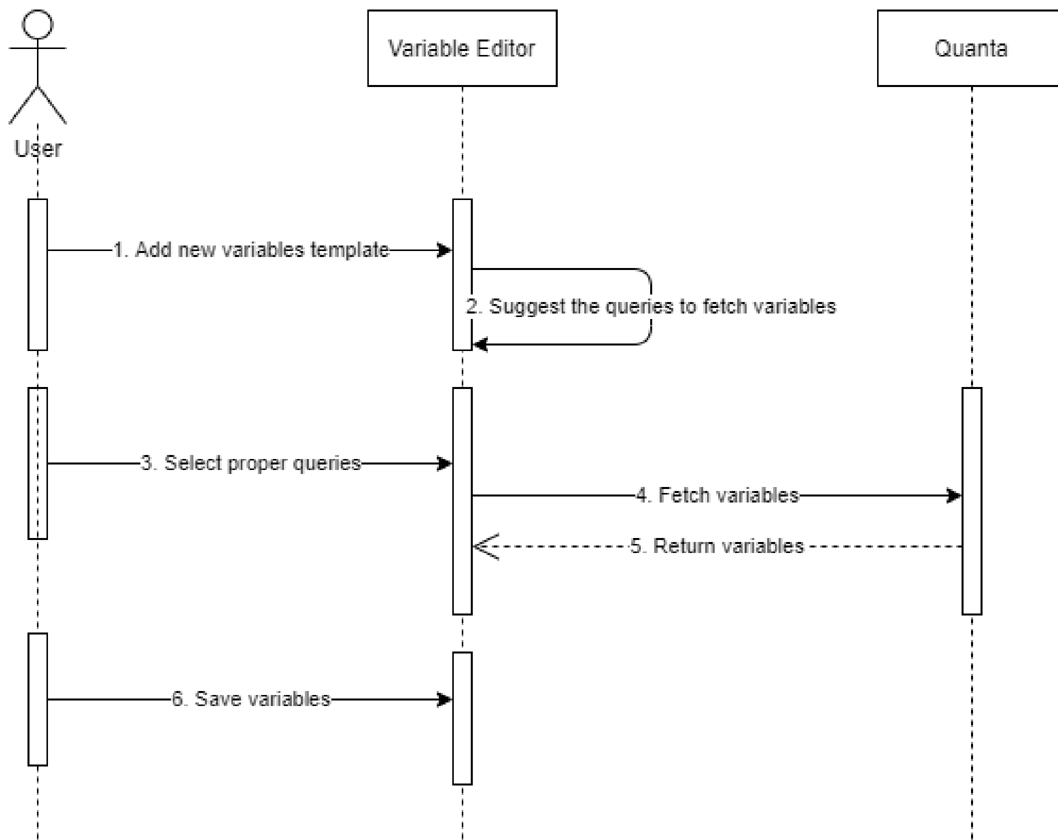

Figure 9. Variable Configuration sequence diagram

The user needs to create a variable template, and the Variable Query Editor will suggest the available queries to fetch the variables option from Quanta. Users can select from the suggestion or create their own variables to suit their needs. After creating the variables, users can save variables for later usage.

4 GUI DESIGN

This chapter explains how the graphical user interface of the application was designed.

4.1 Config Editor

The Config Editor is the entry point for the data source plugin, we need to configure necessary properties to access the Quanta platform. The figure below shows the Config Editor GUI.

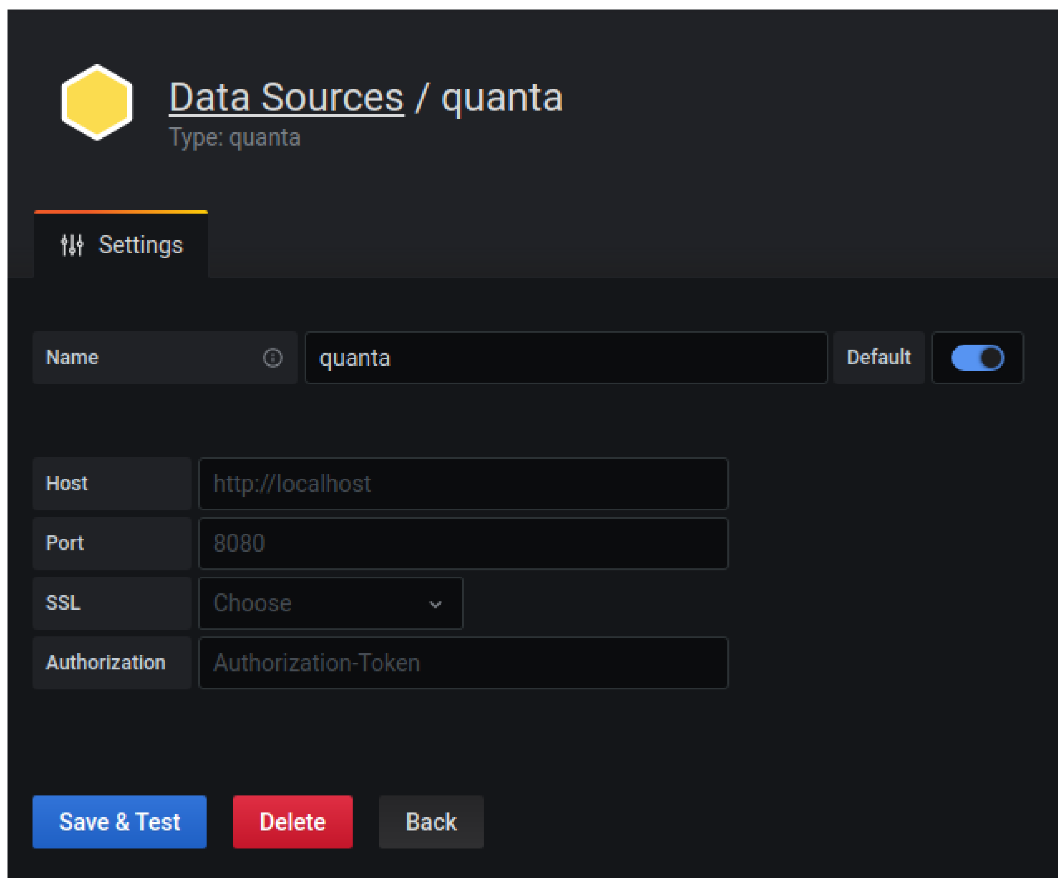

Figure 10. Config Editor GUI

In this Config Editor, users can set the name for the data source plugin. We can have multiple data source plugins with different names. Users need to specify the host, port, SSL flag, and authorization token. The host address is the address of the

Quanta endpoint. The port needs to be numeric. The SSL flag is a dropdown selection with value true or false. The authorization token is a secret form field, which can-not display in the front end for security.

4.2 Query Editor

The Query Editor is where the users create the queries. The figure below shows the Query Editor GUI.

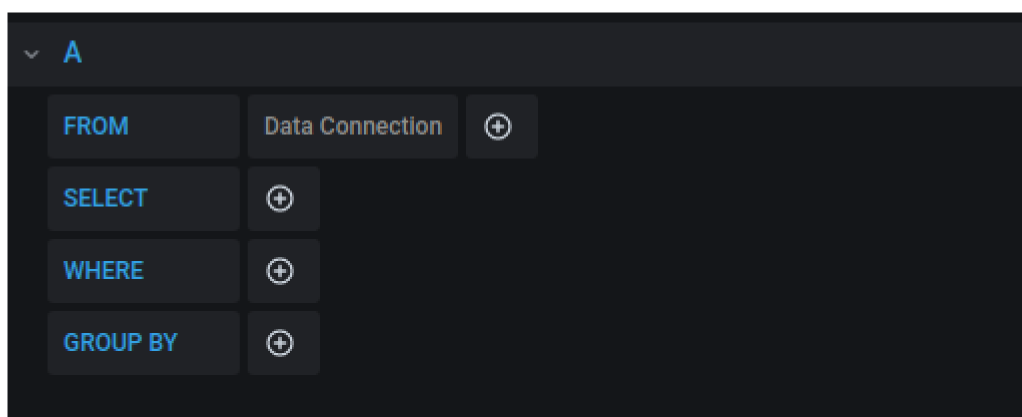

Figure 11. Query Editor GUI

Users can start creating the queries by clicking the plus button. The data source plugin helps the users by providing the available options fetched asynchronously. The users can input their own value if the available options are not suited to their needs.

The query will be divided into following parts:

- **FROM**: where users select Data Connection, Data Series, Task, and Invocation
- **SELECT**: where users select the columns to fetch the data.
- **WHERE**: where users set the condition for the queries
- **GROUP BY**: where users group the value.

Grafana also offers other features such as duplicate the query, copy, export the query.

4.3 Variable Query Editor

The Variable Query Editor is where users can create variables applied to the metrics. The figure below shows the Variable Query Editor GUI.

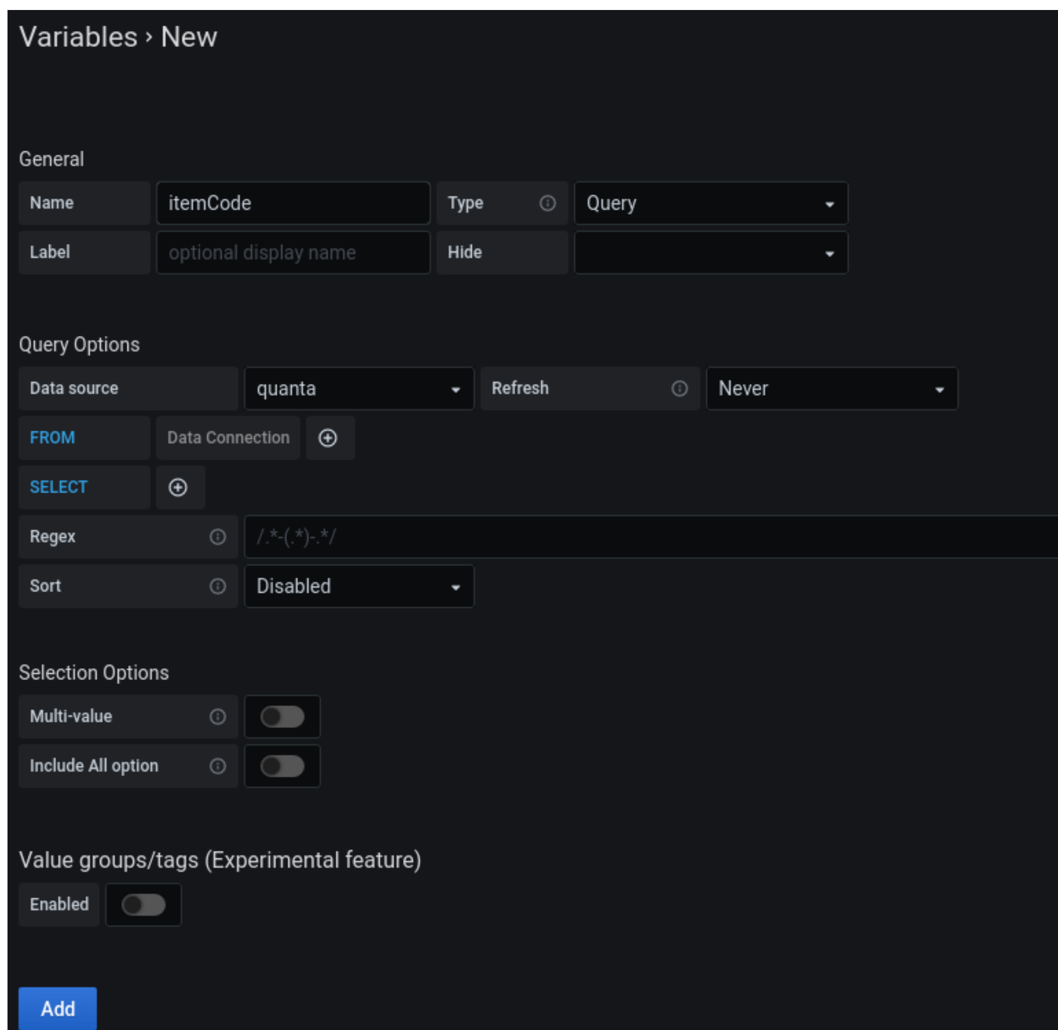

Figure 12. Variable Query Editor GUI

In the Variable Query Editor, users need to fill in the required fields for Grafana to understand the variable template to. The required fields are name, type, and data source. These fields are required by the Grafana variable template. Variable Query

Editor will be rendered when we select the data source, each data source can implement its own Variable Query Editor.

The Regex field is where users specify the regular expression to indicate the search pattern. The sort field is used to sort all the variables. Grafana takes care of Regex and Sort under the hood.

After users have selected all the required fields, they can click the *Save* button to save the variables.

5 IMPLEMENTATION

This chapter explains how the data source plugin was developed. The implementation of each component will be explained.

5.1 Implementation overview structure

Every plugin requires at least two files:

- **plugin.json:** contains information about the plugin
- **module.ts:** exposes the implementation of the plugin

The data source plugin was developed and focused on the following properties:

- **Config Editor**: the configuration where users need to set up the host address, port, SSL flag, and Authorization Token.
- **Query Editor**: constructs queries that return to Data Source to fetch data from Quanta
- **Data Source**: fetches the data from Quanta, map data to Data Frames in which Grafana can recognize the data
- **Variable Query Editor**: defines the variable query, or variable values that return to Data Source to apply variables to all the queries, or specific queries

The figure below shows the workflow of Grafana data source plugin

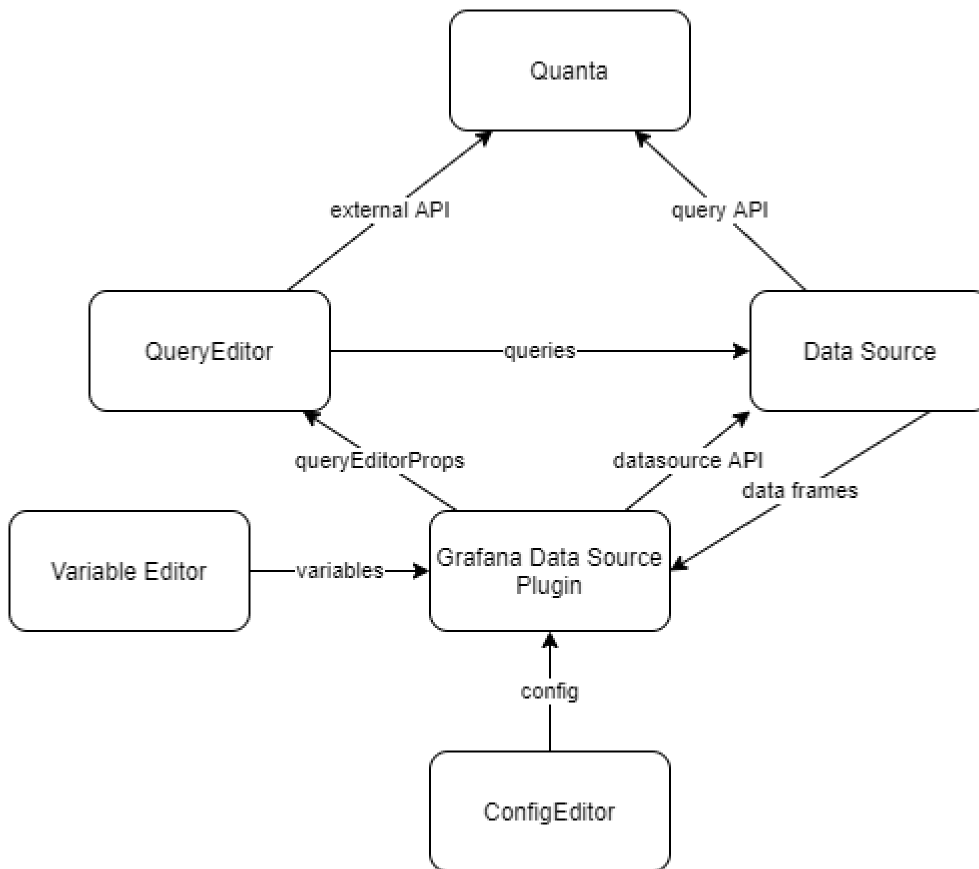

Figure 13. Grafana data source plugin workflows

If the plugin exists, users can find it in Grafana. After adding a data source plugin, users need to specify the host, port, SSL flag, and client-token with the Config Editor. When the configuration is set, Data Source would call the *testDatasource*()**,** implement a health check for the data source and return the message to clients. After adding a data source successfully, users can create a panel and start querying the data with the Query Editor.

After constructing the query in the Query Editor, the data source will query to Quanta via query APIs provided by it to fetch the data and map them to the Data Frames concept of Grafana. Users can create multiple panels with different query and data connections. When users have multiple panels, they can utilize the variables defined at the Dashboard scope with the Variable Query Editor which will apply to all the panels.

5.2 Setting up Environment

There are two ways of setting up the environment: installing Grafana locally or running Grafana using Docker. Installing Grafana can require multiple steps and Grafana may need to be re-installed when we change the local environment. Docker can simplify the installation by running the Docker container images, and we can run Grafana using Docker in any environment that supports it. We can also install Grafana locally with its documentation or running Grafana with Docker in the following steps.

To run Grafana with Docker, we create *docker-compose.yml* file which allows us to deploy, combine and configure multiple docker-containers at the same time. The figure below shows the *docker-compose.yml* file of this project.

```
version: "3"
services:
  grafana:
    image: grafana/grafana:7.3.7
    ports:
      - 3000:3000
    volumes:
      - ./:/var/lib/grafana/plugins
      - grafana-data:/var/lib/grafana
# Explicitly define the persistent volume for your data storage
volumes:
  grafana-data:
    external: true
```

Code Snippet 3. Docker-compose.yml

The *docker-compose.yml* has the following properties:

- **version 3:** using version 3 of Docker Compose

- **services:** defines all of the containers to be created. In this docker-compose file, we have one service which is Grafana
- **grafana:** name of the service
- **image:** running service using a pre-built image, while Docker Compose will fork a container from that image location
- **ports:** maps the container's ports to the host machine
- **volumes:** mounting disks in Docker
- **external:** if set to *true*, specifies that this volume has been created outside of Compose. In this Compose, the external defined the persistent volume for Grafana data storage

After having the *docker-compose.yml* file, we could go to the root folder where the *docker-compose.yml* is stored and run the command:

```
docker-compose up
```

Code Snippet 4. Command to run Docker-compose

This command will create and start the docker container. When Grafana is ready, we can access the Grafana at *http://localhost:3000* with the username by default as *admin* and the password by default as *admin*

To start developing a data source plugin, *grafana-toolkit* which is a CLI application that simplifies all the setup required to develop the plugin is used. The toolkit takes care of building and testing the plugin development. To use *grafana-toolkit*, the local environment needs to install NodeJs and *grafana-toolkit*. Run the following command line:

```
# Create development folder
npx @grafana/toolkit plugin:create quanta-grafana-datasource
# Go to created directory
cd quanta-grafana-datasource
# Install dependencies
```

```
yarn install

yarn dev
```

Code Snippet 5. Create the project folder with Grafana-toolkit

The command above creates the template for the project. After running the command above, we need to restart the Grafana server to discover a new plugin. Open Grafana, go to **Configuration -> Plugins**. Make sure the plugin is there.

5.3 Anatomy of Data Source Plugin

Every plugin requires at least two files: *plugin.json* and *module.ts***.**

The code snippets below show the *plugin.json* file in the data source plugin.

```
{
  "type": "datasource",
  "name": "quanta",
  "id": "jubic-oy-quanta",
  "metrics": true,
  "info": {
    "description": "",
    "author": {
      "name": "Jubic Oy",
      "url": ""
    },
    "keywords": [],
    "logos": {
      "small": "img/logo.svg",
      "large": "img/logo.svg"
    },
    "screenshots": [],
```

```
    "version": "%VERSION%",
    "updated": "%TODAY%"
  },
  "dependencies": {
    "grafanaVersion": "7.3.7",
    "plugins": []
  }
```

Code Snippet 6. plugin.json

When Grafana starts, it will search the plugin directory that contains a *plugin.json* file. Different plugins can have different configuration options. The schema *plugin.json* can be found in the Grafana official documentation. All the plugins require the following properties:

- **type:** Grafana supports three types of plugin: *panel***,** *datasource***,** and *app***.** In this project, the type of plugin is *datasource*
- **name:** name of plugin. In this project, name of plugin is *Quanta*
- **id:** unique identities of plugin. In this project, the id is *jubic-oy-quanta*

The entry point for every plugin is *module.ts*, which exposes the implementation of a plugin. The code snippet below describes the *module.ts* file:

```
export const plugin = new DataSourcePlugin<DataSource, Data-SourcePluginQuery, QuantaDataSourceOptions>(DataSource)
  .setConfigEditor(ConfigEditor)
  .setQueryEditor(QueryEditor)
  .setVariableQueryEditor(VariableQueryEditor);
```

Code Snippet 7. module.ts

The *module.ts* needs to expose an object that extends Grafana Plugin, which can be any of: Panel Plugin, Data Source Plugin, or App Plugin. In this project, we are

using Data Source Plugin. We need to set the ConfigEditor, QueryEditor, and VariableQueryEditor in this *module.ts*.

5.4 Config Editor

Query Editor implemented input fields for host address, port, SSL flag, and Authorization token. To develop the UI, we can use the components in the Grafana SDK.

```
  <div className="gf-form-group">
      <div className="gf-form">
        <FormField
          label="Host"
          labelWidth={6}
          inputWidth={20}
          onChange={onUpdateDatasourceJsonDataOption(props,
'host')}
          value={jsonData.host || ''}
          placeholder="http://localhost"
        />
      </div>

      <div className="gf-form">
        <FormField
          label="Port"
          labelWidth={6}
          inputWidth={20}
          onChange={onUpdateDatasourceJsonDataOption(props,
'port')}
          value={jsonData.port || ''}
          placeholder="8080"
        />
```

```
      </div>


      <div className="gf-form">
        <label
            className="gf-form-label width-6"
        >
            SSL
        </label>
        <Select
          className="width-10"
          options={SSLOptions}
          value={sslValue}
          onChange={(e) => {
            onSSLChange(e.value || false);
            setSSLValue(e);
          }}
        />
      </div>


      <div className="gf-form">
        <SecretFormField
          isConfigured={(secureJsonFields && secureJsonFields.to-
ken) as boolean}
          label="Authorization"
          labelWidth={6}
          inputWidth={20}
          onChange={onUpdateDatasourceSecureJsonDataOption(props,
'token')}
          onReset={onResetAuthorizationToken}
          value={secureJsonData.token || ''}
```

```
            placeholder="Authorization-Token"
          />
        </div>
      </div>
```

Code Snippet 8. Config Editor UI component

All fields are implemented with *<FormField>* except Authorization Token field implemented with *<SecretFormField>* for security. The *onChange* properties of the fields implemented with *<FormField>* will be handled with the function *onUpdateDataSourceJsonDataOption* provided by Grafana SDK. This function takes two arguments: options editor props and the key to prop. With the *<SecretFormField>*, the Authorization Token will be hidden from the front end, the users can-not see, or modify once they set it. They need to reset the Authorization Token and add the new one.

5.5 Query Editor

Each data source has a way to query data using its own query language and Quanta also has its own query definitions. Grafana allows developers to build support for it. To support custom queries, data source plugin implements a Query Editor using React components, which allow users to create their queries. The code snippet below shows how the Query Editor was implemented in the Data Source Plugin.

```
<>

      <FromClause  url={props.datasource.url}  onChange={onChange}
query={query} />


      <SelectClause url={props.datasource.url} onChange={onChange}
query={query} onRunQuery={onRunQuery} />


      <WhereClause  url={props.datasource.url}  onChange={onChange}
query={query} onRunQuery={onRunQuery} />
```

```
    <GroupByClause  url={props.datasource.url}  onChange={on-
Change} query={query} onRunQuery={onRunQuery} />

</>
```

Code Snippet 9. Query Editor component.

Quanta has its own definitions of query API, so that the Query Editor implemented *FROM, SELECT, WHERE,* and *GROUP BY* clauses.

- **FROM**: in this clause, the data connection, data series, task, and invocation will be defined. To implement it, Query Editor implemented the *SegmentAsync* provided by the Grafana development kit for the optimization. *SegmentAsync* will make the asynchronous call, using an external API provided by Quanta to retrieve the necessary data so that the data will only be loaded if it is selected.
- **SELECT**: in this clause, the columns will be selected, and a query constructed by the data retrieved from *FROM* clause. The data source plugin starts querying when a column is selected. This clause implemented *ButtonCascader* provided by the Grafana development kit which is a dropdown selector that users can select the available columns under available data series, or invocations.
- **WHERE**: in this clause, the condition queries will be set. For the columns to specify the data value, users will need to input the value. Other columns could be fetched from Quanta so that the available options will be shown.
- **GROUP BY**: this clause will perform the grouping of value. It works the same as GROUP BY statement in any query language.

All the queries from those clauses will be in the format that Quanta query API recognizes, and the Grafana data source plugin will make the external API call Quanta via query API.

5.6 Data Source

Every data source plugin must extend the *DataSourceApi* interface, for it must implement two methods: *query* and *testDataSource*. *DataSouceApi* interface provides various other methods to implement different features, which are described in Grafana official document. This project focused on three methods: *query*, *testDatasource*, and *metricFindQuery*

query() is the core of any data source plugin where the data is returned in Grafana's data format by accepting a query from the user, retrieving data from an external database. Queries defined at the Query Editor will be returned to this method as an argument along with context information. The code snippet below shows how the *query()* method was implemented.

```
async query(options: DataQueryRequest<DataSourcePluginQuery>): Promise<DataQueryResponse> {

    const promises = options.targets.map((query) => {

      if (!query.selectors || query.selectors.length === 0) {

        return [];

      }


      const selectorsWithVariables =

        getTemplateSrv().getVariables().length === 0

          ? query.selectors

          : query.selectors.map((selector) => {

              if (selector?.match(VARIABLE_REGEX)) {

                return getTemplateSrv().replace(selector, options.scopedVars);

              }

              return selector;

            });
```

```
      const newQuery = {
        interval: query.interval && query.interval !== 0 ? query.interval : options.intervalMs / 1000,
        selectors: selectorsWithVariables,
        start: moment.utc(options.range.from.toString()).format(API_DATE_FORMAT) + 'Z',
        end: moment.utc(options.range.to.toString()).format(API_DATE_FORMAT) + 'Z',
      };

      return client.queryTimeSeries(this.url, newQuery).then((res) => {
        return res.reduce(
          (frames, queryResult) => [
            ...frames,
            ...Object.entries(queryResult.measurements[0].values)
              .filter((value) => filterOutGrouping(value[0]))
              .map((entry, i) => {
                const frame = new MutableDataFrame({
                  refId: query.refId,
                  fields: [
                    { name: 'Time', type: FieldType.time },
                    {
                      name: `${entry[0]}${getGroupingSuffix(queryResult.measurements[0].values)}`,
                      type: FieldType.number,
                    },
                  ],
                });

                queryResult.measurements.forEach((measurement) => {
```

```
                  const  time  =  moment.utc(measurement.time).for-
mat(INPUT_DATE_FORMAT);

                  const  value  =  Object.entries(measurement.val-
ues).filter((value) => filterOutGrouping(value[0]))[i];

                  frame.appendRow([time, value[1]]);

                });


                return frame;

              }),

          ],

          [] as MutableDataFrame[]

        );

      });

    });


    return Promise.all(promises).then((data) => {

      return { data: data.reduce((a, b) => [...a, ...b], []) };

    });

  }
```

Code Snippet 10. query() method

In this *query()* method, the queries return from the Query Editor, and will be added with the time range, and the time interval before querying with Quanta APIs. When the queries are converted to a format that Quanta can understand, this method will run the *queryTimeSeries* Quanta API to fetch the data, and the data will be mapped to Data Frames in which Grafana can recognize. Data Frames is a collection of fields to map data to Grafana data formats and each of them consists of values along with meta information of the field. After querying, the data frames are sent to Grafana to visualize the data.

testDatasource() is required to check the connection by implementing a health check for the data source. The code snippet below shows how the *testData-Source()* was implemented.

```
async testDatasource() {
    const response = await client.testDataConnection(this.url);
    if (response.ok) {
      return {
        status: 'success',
        message: 'Success',
      };
    }

    return {
      status: 'failed',
      message:   `${response.status.toString()}:   ${response.sta-
tusText}`,
    };
}
```

Code Snippet 11. testDatasource() method

The method *testDatasource()* will call the *testDataConnection* Quanta API to implement the checking for the data connection. If the connection is valid, it will return the status success, otherwise it will return status failure with the error message.

metricFindQuery() defined at the Variable Query Editor will be returned to this method as an argument. This method executes the variable queries and returns variables in a format that Grafana recognizes. The code snippet below shows the implementation of this method.

```
async metricFindQuery(query: DataSourcePluginQuery, options: any)
{

    if (!query.selectors || query.selectors.length === 0) {

      return [];

    }



    const newQuery = {

      interval: DEFAULT_TIME_INTERVAL,

      selectors: query.selectors?.map((selector) =>   `dis-
tinct(${selector})`),

    };



    return client.queryTimeSeries(this.url, newQuery)

      .then((res) => {

      return res

        .map(

      (queryResults) =>

          queryResults.measurements

            .reduce(

            (metricValues, measurement) => [

              ...metricValues,

                    ...Object.entries(measurement.val-
      ues).map((pair) => ({

                text: pair[1],

                value: pair[1],

              })),

            ],

            [] as MetricFindValue[]

          )

        )
```

```
        .reduce((a, b) => [...a, ...b], []);
    });
}
```

Code Snippet 12. metricFindQuery() method

This method will query distinct values from the Quanta backend to list the available options for the suggestion. Users can define their own variables to suit their needs.

5.7 Variable Query Editor

Variables are placeholders for values and can be used for metric queries and panel titles. As the variable changes, the metric query of the panel on the Dashboard will also change to reflect the new value. With the variables, the queries become more flexible, and the variables can be applied to any queries following the template. There are various format options and in this data source plugin, the syntax for variable is: *$variable_name*. The code snippet below shows how the Variable Query Editor was implemented

```
<>
      <FromClause    url={datasource.url}    onChange={saveQuery}
query={query} />
      <SelectClause   url={datasource.url}   onChange={saveQuery}
query={query} onRunQuery={() => refresh} variable />
 </>
```

Code Snippet 13. Variable Query Editor component

The Variable Query Editor reuses the *<FromClause>* and the *<SelectClause>* which are used to fetch the distinct options from the Quanta backend for the suggestion. After selecting the variable queries, they will be returned to the Data Source to fetch the data from Quanta. If the variables are not available in the Quanta backend, users can define their own variables based on the Grafana variable template.

6 TESTING THE DATA SOURCE PLUGIN

This section guides users to test the data source plugin step by step with the local environment. We assume that the use case of testing is when we have a data set of sales data, which is a CSV files with three fields:

- **time**: Date time for the sales value of an item.
- **value**: Sales value of an item in a specific date time.
- **itemCode**: Item code

A customer would like Quanta to run the data analysis to forecast the future sales value and visualize the result on the Grafana data source plugin. Before using the Grafana data source plugin, we need to add data connection in Quanta, set the task for the Workers to run the algorithm to analyze, and forecast the sales value in the future. After Quanta runs their algorithms, and returns the data to the database, we can start using the Grafana data source plugin to visualize the data.

Firstly, the configuration for the data source plugin is set up as Figure 14.

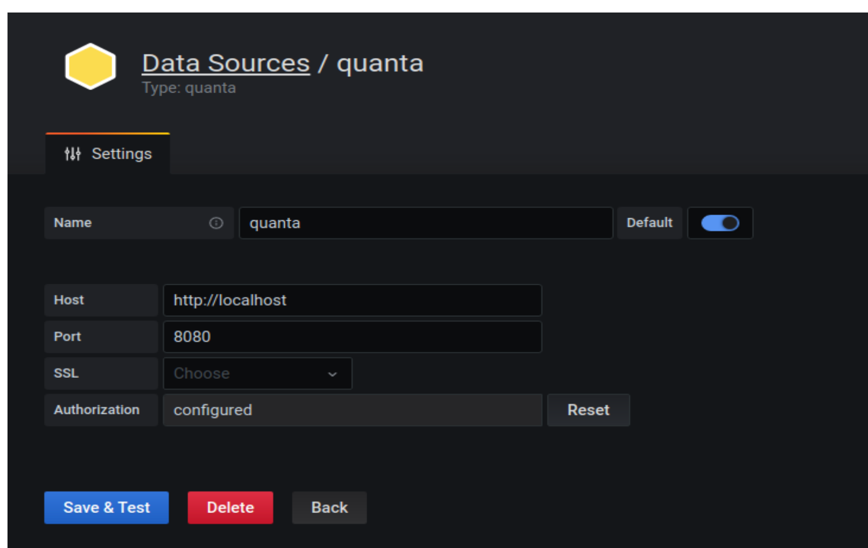

Figure 14. Data source configuration example

This data source plugin is tested with a local environment, so the host will be *http://localhost* and the port is *8080*. The authorization token is generated by Quanta, and we can get the token in the Quanta platform.

After configuring the necessary properties, go to *Dashboard -> Add new panel* and start querying. In the query editor, we are going to query the data after ingesting and formatting by Quanta called *series*, and the forecasted values called *result_output* data.

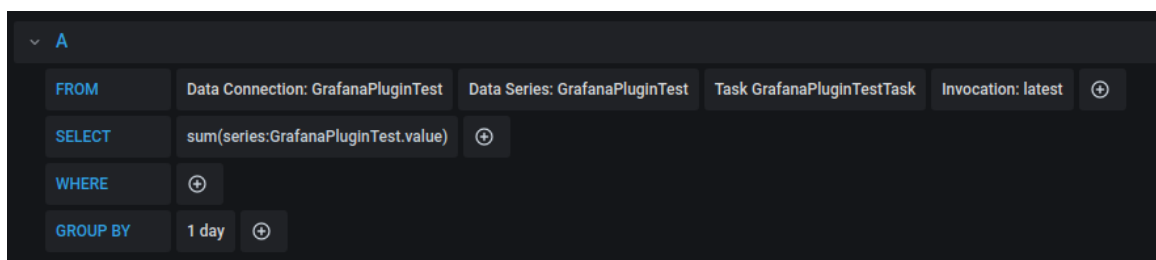

Figure 15. Series data query example

The Figure 15 above shows the example of the query. In this query, we have the data connection named *GrafanaPluginTest*, and data series as *GrafanaPluginTest* the task that assigns the necessary information to make the worker understand, and the invocation as the snapshot of task information, which is immutable and passes to the worker for running the analyzation. Then, we select the column to visualize in this example, the column is *value*, which is the sales data at a specific date time. We will select the sum of sales value, and group by 1 day of data that means selecting the sum of sales value in 1 day. The data will be visualized as the figure below.

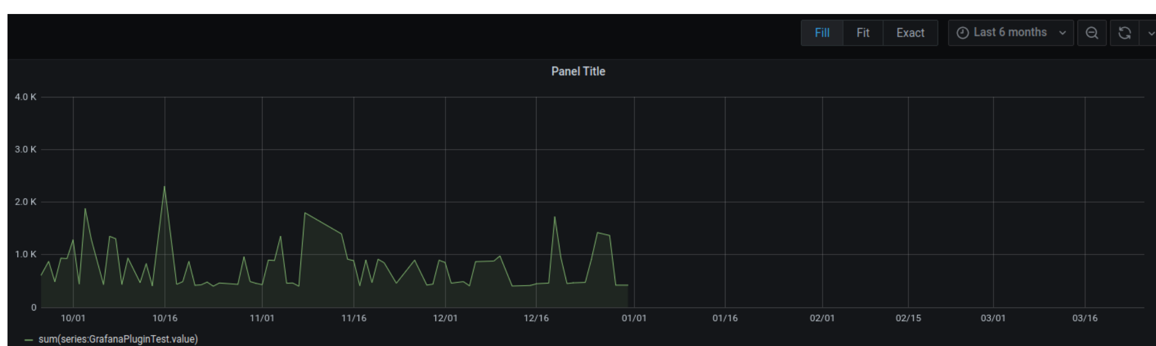

Figure 16. Series data visualization example

We can add the forecast value in the same panel with the series data by adding one more query for the forecasted value as shown in Figure 17 below.

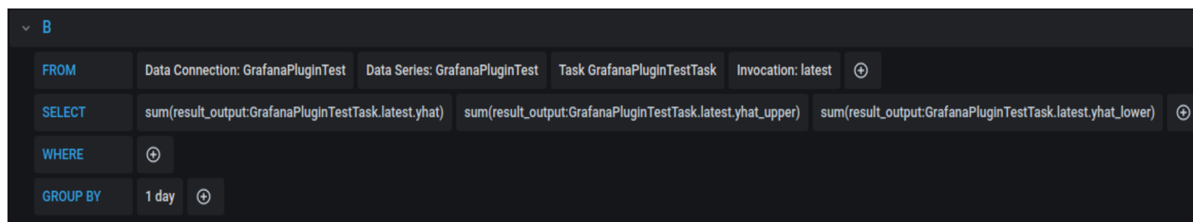

Figure 17. Result output data queries example

The column *yhat* is the worker definition for the forecast value. The worker returns *yhat_upper* for upper bound, *yhat_lower* for lower bound so that users can see the range of forecasted values. The Figure 18 below shows the data visualized in the same panel.

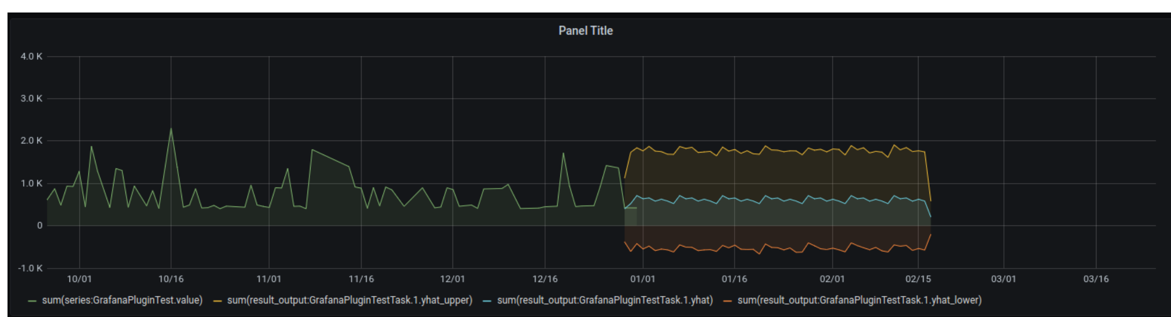

Figure 18. Series data and the forecasted value in the same panel

If users want to visualize exactly one item code, the item code will be applied to all the queries to filter that item code. The variables perfectly fit this situation. Go to the *Dashboard -> Setting -> Add a variable*, then define the variable. The variable will be displayed at the top of the panel as seen in figure below.

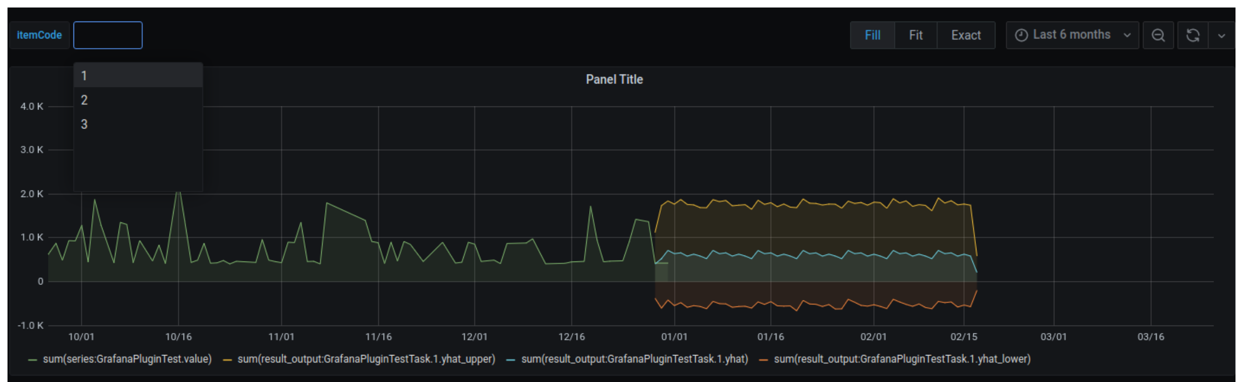

Figure 19. Variables in the panel example

In this example, we have three item codes: 1, 2, and 3. To use the variable, the query must follow the syntax that the *Query Editor* can understand. The *WHERE* clause is where the conditions are set, the variable will apply to any query that has this syntax: *$variable_name*. The figure below shows the variable example.

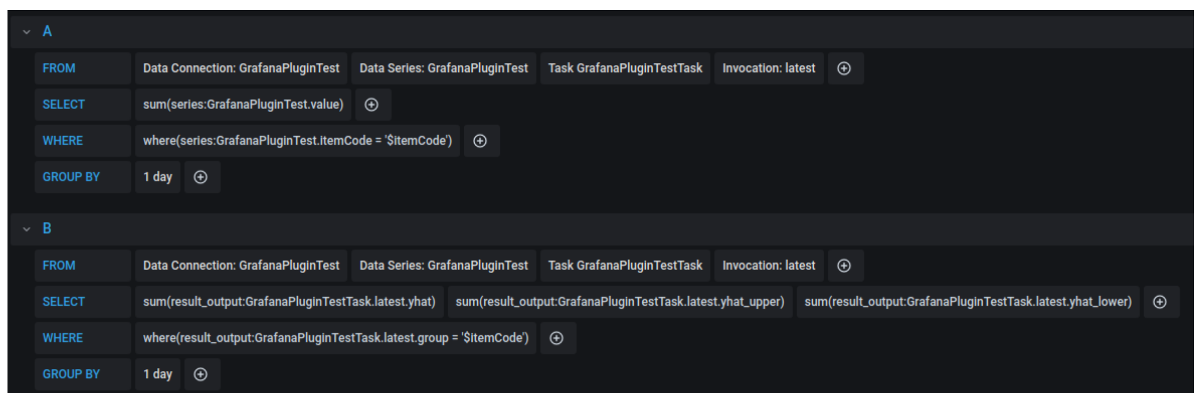

Figure 20. Variable syntax example

After querying, the panel visualized exactly with the item code selected in the variables as in Figure 21 below.

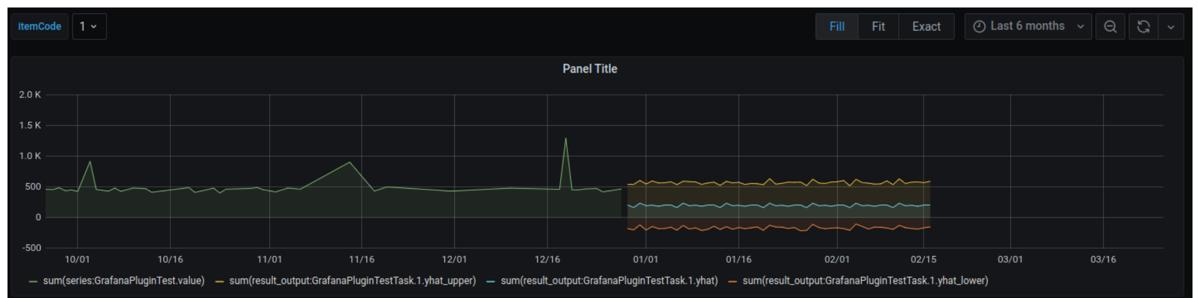

Figure 21. Data visualization with the variables

The variables are not restricted to a specific panel thus we can apply the variables to multiple panels in the same dashboard. In the dashboard, we can have multiple panels to query various types of data and display in various charts as in the figure below.

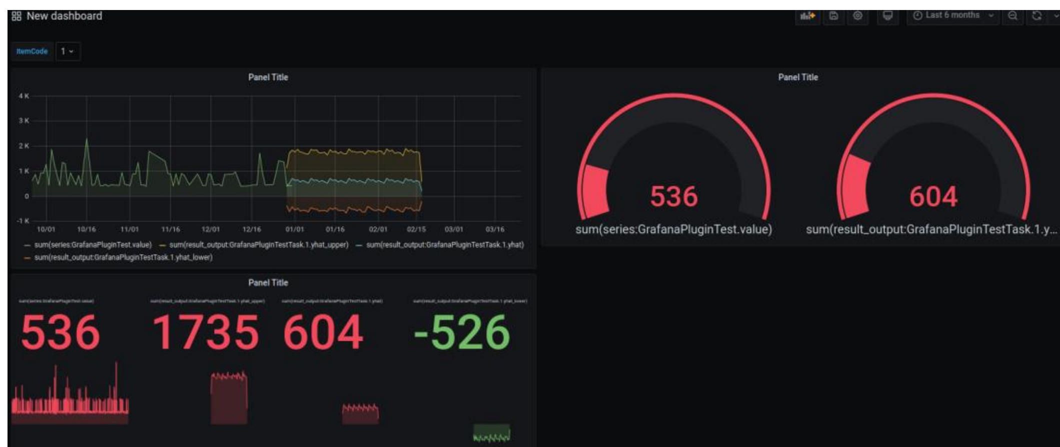

Figure 22. Dashboard with multiple panels

7 CONCLUSION

The aim of the thesis project was to develop a Grafana data source plugin integrated with the Quanta platform, which can fetch the data from Quanta to visualize Grafana. To achieve these goals, the data source plugin is developed with the Grafana SDK, ReactJs, and TypeScript. The application has already successfully implemented the core functionalities, which are able to fetch data from Quanta and visualize data fetched on the Grafana.

The most challenging part of this project was building the Data Source that could recognize the data fetched from Quanta. The data fetched from Quanta can come with different shapes and types, the Data Source needs to process this data and map the data to Data Frames concepts that Grafana can understand. After recognizing data format, Grafana can start visualizing the received data.

This project was used in the testing phase and received positive feedback from Product Owner Jubic Oy.

To summarize, this project achieved its objectives set at the beginning. After the demand analysis, detailed design, coding and testing, the application was completed as scheduled.

7.1 Future Work

This project has potential to continue developing, Grafana provides a considerable number of features. We can define the threshold for the data, and develop the Alert, which notify the user via Slack, PagerDuty, and more. With the Alert, we can control the anomaly data by sending the notification to the platform we select. Grafana also allows mixed data sources in the same graph, so we can combine multiple data sources in the same panel.

REFERENCES

/1/ Quanta. Quanta Github Repository. Accessed 21/03/2021

https://github.com/jubicoy/quanta

/2/ Grafana. Grafana Official Website. Accessed 20/03/2021

https://grafana.com/grafana

/3/ Grafana. Wikipedia. Accessed 21/03/2021

https://en.wikipedia.org/wiki/Grafana

/4/ JavaScript. Wikipedia. Accessed 20/03/2021.

https://en.wikipedia.org/wiki/JavaScript

/5/ React (JavaScript library). Wikipedia. Accessed 20/03/2021

https://en.wikipedia.org/wiki/React_(JavaScript_library)

/6/ React. React Official Website. Accessed 20/03/2021

https://reactjs.org/

/7/ TypeScript. Wikipedia. Accessed 20/03/2021.

https://en.wikipedia.org/wiki/TypeScript

/8/ TypeScript. Typescript Official Website. Accessed 20/03/2021

https://www.typescriptlang.org/

/9/ TimescaleDB. Github. Accessed 20/03/2021.

https://github.com/timescale/timescaledb

/10/ Representational state transfer. Wikipedia. Accessed 20/03/2021

https://en.wikipedia.org/wiki/Representational_state_transfer

/11/ Docker overview. Docker official website. Accessed 20/03/2021

https://docs.docker.com/get-started/overview/

/12/ Docker-hub. Docker official website. Accessed 20/03/2021

https://www.docker.com/products/docker-hub

/13/ Reading data. TimescaleDB official website. Accessed 31/04/2021

https://docs.timescale.com/latest/using-timescaledb/reading-data

/14/ TimescaleDB. TimescaleDB official website. Accessed 31/04/2021

https://www.timescale.com/

/15/ Features. Grafana official website. Accessed 31/04/2021

https://grafana.com/grafana/

/16/ How RESTful APIs work. Accessed 31/04/2021

https://searchapparchitecture.techtarget.com/definition/RESTful-API

/17/ Plugins. Grafana Document. Accessed 31/04/2021

https://grafana.com/docs/grafana/latest/plugins/

/18/ Data sources. Grafana plugins. Accessed 31/04/2021

https://grafana.com/grafana/plugins/

/19/ Apps. Grafana plugins. Accessed 31/04/2021

https://grafana.com/grafana/plugins/

/20/ Panels. Grafana plugins. Accessed 31/04/2021

https://grafana.com/grafana/plugins/