Expertise
and insight
for the future

Arsenii Kurilov

Developing a Reporting Web Application for Factory Production Efficiency Analysis

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

13.02.2018

Author(s) Title	Arsenii Kurilov Developing a Reporting Web Application for Factory Production Efficiency Analysis
Number of Pages Date	43 pages 13 Feb 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialization option	Software Engineering
Instructor(s)	Hannu Markkanen, Research lecturer Petri Uusitalo, Manager R&D, Finn-Power Oy

The goal of the thesis is to investigate software development methods and techniques used in the implementation of a web-based reporting application. The paper discusses in detail topics such as data source and data store management, data aggregation and data visualization in the context of the development of a real reporting software product for a company. The paper mainly focuses on the explanation of data aggregation techniques and usage of custom data search API.

The thesis gives a general overview of common architecture and technologies used in business intelligence applications and briefly introduces several existing business intelligence solutions. It also discusses technologies that can be used for implementing such solutions.

Most of the thesis is dedicated to the explanation of the development process of a web-based reporting application with .NET back-end and React client. These and other technologies used in the implementation of the project are briefly presented in the paper and the motivation behind the choice of the technology stack is explained. The paper provides implementation details of the data source layer of the reporting application and detailed examples of the application's data aggregation and visualization logic capabilities.

Keywords	reporting, business intelligence, web-application, .NET, React, data visualization

Contents

List of Abbreviations

API	Application programming interface
ASCII	American Standard Code for Information Interchange
CRUD	Create, read, update and delete
ELK	Elasticsearch, Logstash, Kibana
ERP	Enterprise resource planning
GUI	Graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoC	Inversion of control
JSON	JavaScript Object Notation
MSSQL	Microsoft Structured Query Language
ORM	Object-relational mapping
RDBMS	Relational database management system
REST	Representational State Transfer

1 Introduction

Today business intelligence (BI) solutions are used in many business sectors. Some examples of sectors where BI solutions are applied are port industry [1], banking sector [2], retail [3]. BI applications are used for analyzing existing data collected by the customer and deriving business benefits from such analysis, such as more efficient decision

making, improvement of financial performance, increasing business competitive possibilities [1, 2].

The project described in this paper is a business reporting application, including some elements of BI, and it is being developed for Prima Power. Prima Power designs and manufactures machines for sheet metal working, provides service for those machines and develops a variety of software related to machine operation and other customer needs, such as business reporting.

The thesis describes a web application further in text referred to as Reporting Application. Reporting Application retrieves data from historical data sources, provides flexible web API and data visualization functionality. Reporting Application uses the well-established client-server model, and it is being developed with .NET back-end, React.js front-end and Microsoft SQL (MSSQL) server used for data persistence. The goals of this thesis were to investigate software development methods and techniques used in the implementation of a web-based reporting application.

2 Project goals

The main goal of Reporting Application is to replace the existing business reporting solution, developed previously by the company. The existing solution is a Windows desktop application, providing end-user functionality similar to the functionality of Reporting Application.

However, the existing software has two major drawbacks. Firstly, it is not a distributed solution and a separate installation is required for use on each PC. Secondly, it retrieves data from historical data sources directly, which introduces a set of challenges when it comes to data handling. Reporting Application, on the other hand, is a web-based solution, which means that it uses a distributed application structure and can be accessed from any PC with a web-browser installed. Reporting Application retrieves historical data and stores it in a dedicated MSSQL database, this ensures that functionality of the application will not be interrupted if one of the data sources will go offline.

Reporting Application uses a set of up-to-date web technologies, such as .NET Web API, Entity Framework and MSSQL database for the server side and React.js for client side. It also provides a flexible web API, inspired by Elasticsearch. The flexibility of the API

makes it possible to extend the client functionality and to introduce new data visualization in a faster and more efficient way.

3 Common architecture and technologies used in business intelligence applications

Reporting Application introduces some elements of Business Intelligence. Prior to the development of the application, some of the modern BI solutions, existing on the market, were examined. This helped to make better decisions in questions of application architecture and functionality.

The main purpose of any BI system is to analyze volumes of collected business operation related data and to present this data to the user in a useful and meaningful way. Thus, BI systems usually have the following application levels: data source layer, data processing layer and interaction or representation layer [1, 2, 3]. The layers listed above can differ in complexity depending on the solution requirements, but this structure is quite common for many BI solutions.

BI solutions can be implemented by using various programming languages, software frameworks, and data persistence methods. The choice of technologies depends on specific project requirements. The choice of operation environment depends on customer needs as well. Today cloud-based solutions are gaining popularity [4], but in some cases, for example, because of security reasons, on-premises solutions are preferred.

3.1 The common architecture used in BI applications

BI applications usually share a common architecture pattern. Though there is no consistent scheme behind naming the architecture layers that usually compose a BI solution, conceptually many BI applications share the same architecture scheme. The architecture is driven by the set of tasks that BI applications should handle, such as accessing collected data, transforming and organizing the data in a meaningful way and providing ways to visualize the transformed data and possibility for the user to interact with the data.

3.1.1 Data source and data management layers

In many cases, the customer might collect and store their operational data in different formats and by means of different software. The variety of data sources might include spreadsheets, databases or special software products such as enterprise resource planning (ERP) systems. [4] The collection of different data sources composes the data source layer.

One of the most important tasks that a BI application should handle is the consolidation of customer data, which means collecting data from different data sources, converting it to a specific format and storing it in a single data store. The consolidated data should be stable and uniform to make possible further data analysis and visualization. [3]

The data access layer handles the connection to the different data source and data retrieval from those sources, consolidation and in many cases, the persistence of the retrieved data. The data access layer should also provide a means for the application service layer to query the consolidated customer data.

3.1.2 Application service layer

The application service layer usually consists of various applications that access data from the data access layer and handle different tasks such as data analysis, data mining, and report generation. In other words, this layer is associated with transforming the customer data into meaningful business information.

Data analysis services provide the possibility to perform flexible and efficient data queries to get meaningful information helpful in making business decisions. Data mining services are based on artificial intelligence and provide a different set of automatic data analysis functions such as data categorizing, data clustering, defining association rules and forecasting. [2] The application service layer can also include services that handle tasks such as report generation and management.

3.1.3 Presentation layer

The presentation or information interaction layer consumes services provided by the application service layer and handles tasks such as data visualization and user interaction. It can present data to the user in different forms such as interactive dashboards or fixed reports containing various data visualizations.

It also provides ways for the user to interact with the data in various ways, such as applying different filters, writing easy-to-compose data queries or defining custom data visualizations.

3.2 Overview of technologies used for implementing BI applications

Usage of implementation technologies depends on customer needs and project scale. If the customer case demands a large-scale solution and extraction of data from a wide variety of data sources, a better approach would be to use an existing BI suite such as Microsoft Power BI or Qlik as the base for the final solution. On the other hand, if the scale of the project is not very large, the number of data sources to support is small and there are technical limitations, a custom solution could be implemented from scratch.

3.2.1 Data management layer technologies

The customer might use different kinds of methods to collect and store the data. The goal of a BI system, however, is to unify and consolidate this data, process also known as data extraction and integration. This task is usually handled by software called data source connector. Data source connector retrieves data from the data source, transforms it and provides it to the data store. Implementation details of a data source connector depend on the nature of a data source itself and on the technology chosen for implementing the data store.

Usually, BI software provides a set of data source connectors out of the box. For example, Microsoft Power BI provides connectors for many kinds of data sources, such as most popular SQL database engines, different formats of spreadsheet files and a broad variety of online data sources such as Microsoft Azure, Salesforce, Google Analytics, and many others. [5]

Qlik Sense provides a similar set of data connectors as well, including most popular SQL databases, many online sources such as Facebook, GitHub, Youtube, as well as standard email protocol connectors and many others. [6]

Elastic's Logstash, a part of Elasticsearch, Logstash, Kibana (ELK) stack also used for implementing BI solutions, supports a wide variety of data sources too. For examples, most popular SQL database engines, several online data sources such as Salesforce and GitHub and many others. [7] For persisting the consolidated data Elasticsearch search engine can be used.

In a custom solution, data source connectors can be implemented from the ground up by using a variety of technologies. For example, Java and JDBC driver can be used, if support for an SQL database engine data source is required. Another example of a data source can be plain text files, containing data in a specific format. In this case, a data source connector can use a set of custom file parsers to retrieve the data from the files. A dedicated SQL database can be used as a data store for persisting the extracted data.

3.2.2 Application service layer technologies

Application services are used to access the consolidated data and to perform a variety of tasks, such as data mining, data analysis and generation of reports.

In Microsoft Power BI and Qlik, a set of proprietary technologies is used for implementing this architecture layer. In ELK stack, Elasticsearch not only acts on the level of data management but also provides powerful functionality for data analysis and report generation as it has very flexible data query API.

Application service layer can be implemented as well from scratch depending on customer needs. In this case, custom application logic can be written in any major programming language, such as Java or C#. The functionality of the layer totally depends on specific case requirements. An efficient and flexible API for querying data from the data warehouse, however, is a necessity for implementing a well-functioning presentation layer of the application.

3.2.3 Presentation layer technologies

The presentation layer should provide ways for the user to read the data in an easy way, for example, by presenting data in forms of different visualizations, and to interact with the data, for example, by writing queries or building custom data visualizations.

The presentation layer functionality is usually provided by a client application. It can be a desktop, mobile or web application. Microsoft Power BI and Qlik provide a range of client applications out of the box (desktop, mobile, and web). Power BI provides a way to develop and integrate custom data visualizations by using D3.js. Qlik uses a custom Picasso.js (based on D3.js) library for rendering the data visualizations.

In ELK stack, the Kibana visualization plugin provides presentation layer functionality. Kibana allows the user to define dashboards containing various data visualizations and uses D3.js for rendering purposes.

In a custom solution, a React.js client application could be developed to function on the presentation layer. A JavaScript-based charting library, such as C3.js, could be used for rendering the data visualizations.

4 Overview of technologies used in Reporting Application

This chapter describes the nature of the data sources, from which historical data should be extracted. It also explains the choice of the relational database management system (RDBMS), frameworks, and libraries used for implementing the server and client side of Reporting Application and gives a brief overview of the chosen technologies.

4.1 Data source layer

Sheet metal work machines generate various production data, including the information about the number of parts produced and their properties and the information about metal sheets and material used for producing those parts. Production order related data is also being generated by using data coming from proprietary software and data coming from the machines. Data from the machines is stored by using two different methods: MSSQL server database for newer installations and plain text ASCII files for the older ones. Part of the production orders related data is collected by using proprietary software, and it is

not yet accessible. It is planned to access this data by using a REST service in the future, thus only MSSQL server database and ASCII files data sources will be described in detail in this paper.

4.1.1 MSSQL server database

Historically, the company focuses on the usage of Microsoft technologies for implementing data collection routines and end-user applications. Factory automation cell controls always have a Windows operating system installed. The choice of MSSQL server, as the RDBMS to be used for the project, is explained by such advantages, as easier security management with Windows authentication and less complicated installation process in a Windows environment.

An MSSQL server database with name HistoryDB is used for storing production data generated by the machines. It includes part production and material consumption data for two types of machines: bending and blanking, data about alarms generated by the machines and machine runtime and feed rate data.

A custom MSSQL server connector is used for extracting data from the HistoryDB. The implementation of the connector is described in detail in Chapter 5.1.

4.1.2 ASCII files

In addition to the MSSQL server database, in older installations, plain text ASCII files are used for storing the data generated by the machines. ASCII files with different formatting and naming schemes are used depending on the data stored.

A custom ASCII files connector, relying on a set of file parsers, is used for extracting data from this data source format. The implementation of this data source connector type is explained in detail in Chapter 5.1.

4.2 Back-end technology stack

The section explains the choice of the technologies used for the implementation of Reporting Application back-end. MSSQL server, .NET Entity Framework, and ASP.NET Web API framework are the core technologies used for implementing the server side of the project.

4.2.1 Dedicated reporting database

Reporting Application uses the MSSQL server, operational database management system developed by Microsoft [8], and a dedicated SQL database, as the technologies used for implementing the data store, containing the consolidated history data. MSSQL server was chosen for the following reasons: there is an instance of MSSQL server usually installed on a customer's site, the developers in the company are familiar with this technology, MSSQL server facilitates security concerns by using Windows authentication.

The previous version of reporting software was reading the data from the data sources directly, however, a different approach was chosen for implementing Reporting Application. The decision was made to implement a dedicated database for storing consolidated data. A dedicated data store has a set of advantages. Firstly, Reporting Application can function, even if the data sources go offline, and it is able to export data from the data sources when they come back online again. The data store can be set up both locally and in the cloud. Data from the data sources can be transformed and enhanced before it is stored in the data store, which facilitates the development of other application functions, such as data search and aggregation and data visualization. The approach with a dedicated database also allows usage of the object-relational mapping (ORM) technique, which facilitates the development and maintenance of the data store.

4.2.2 Microsoft .NET Entity Framework

Entity Framework is an object-relational mapper for .NET. Entity Framework provides features, such as automatic change tracking, eager, lazy and explicit loading, translation of strongly-typed queries using LINQ, both code-first and database-first approaches for creating entity models. [9]

There is a number of reasons behind the choice of Entity Framework as the ORM framework for the project. It is an open source technology, developed and maintained by Microsoft. Consulting services are available to the company regarding this technology. Entity Framework offers support for migrations, which facilitates the development of the data store. It also offers Windows authentication support, which facilitates security implementation.

4.2.3 Microsoft ASP.NET Web API framework

ASP.NET Web API is an open source web API framework, developed and supported by Microsoft. In this project .NET Web API was used for implementing the REST API for querying data and managing various user settings, as well as hosting the client React application.

The factors, such as familiarity of other developers in the company with this framework, easy integration with other .NET technologies and reliable support provided by the framework developer, determined the choice of ASP.NET Web API as the web API framework for this project. In Reporting Application, ASP.NET Web API is hosted with OWIN to remove the dependency on Microsoft IIS and to facilitate the installation of the application in different environments.

4.3 Front-end technology stack

The client side of Reporting Application is implemented with React – Redux stack. C3.js and Vis.js libraries are used for rendering various data visualizations.

4.3.1 React and Redux libraries

React is a JavaScript library for building user interfaces. It offers a declarative, component-based approach for implementing a graphical user interface (GUI), which helps to maintain the codebase well organized. [10] At the moment, React is the leading technology when it comes to implementing web application clients. React is well supported by the open source community and Facebook. Because of the factors listed above React was chosen for implementing the GUI of the client side for Reporting Application.

Reporting Application front-end deals with data handling as well, it performs a large number of search queries and has to store a relatively large amount of data to back up the data visualizations. To maintain the state of the client application in an efficient manner, the Redux library was chosen. Redux is a predictable state container for JavaScript applications [11], and it is widely used in conjunction with React.

4.3.2 C3.js and Vis.js data visualization libraries

C3.js is a charting library based on D3.js [12], it was chosen for implementing data visualization in this project for the following reasons. It offers more high-level API compared

to other D3.js based libraries, such as NVD3, but at the same time offers in-depth possibilities for customization by using D3.js API. C3.js uses SVG for rendering the graphics, which is slower than WebGL used in Chart.js but offers better customization options with CSS styling.

Another visualization library used on the client side is Vis.js. Vis.js is a dynamic JavaScript visualization library [13] Vis.js was chosen because it offers a unique timeline visualization component necessary for alarm and machine state data visualizations.

5 Implementation

This chapter explains the implementation details of Reporting Application. Reporting Application is a web application, using client-server architecture. The client is implemented with React and provides a set of data visualization views and a settings view. The server, implemented by using the .NET framework and C# programming language, uses a dedicated MSSQL server database to store consolidated data and various user settings. The server uses a set of aggregators and flexible data search API to provide data necessary to create client-side data visualizations. It also provides more traditional REST API to handle creation, update, and retrieval of various user settings. The server relies on a set of data source connectors to retrieve data from data sources of two types. The architecture of the application is explained in the diagram displayed in Figure 1.

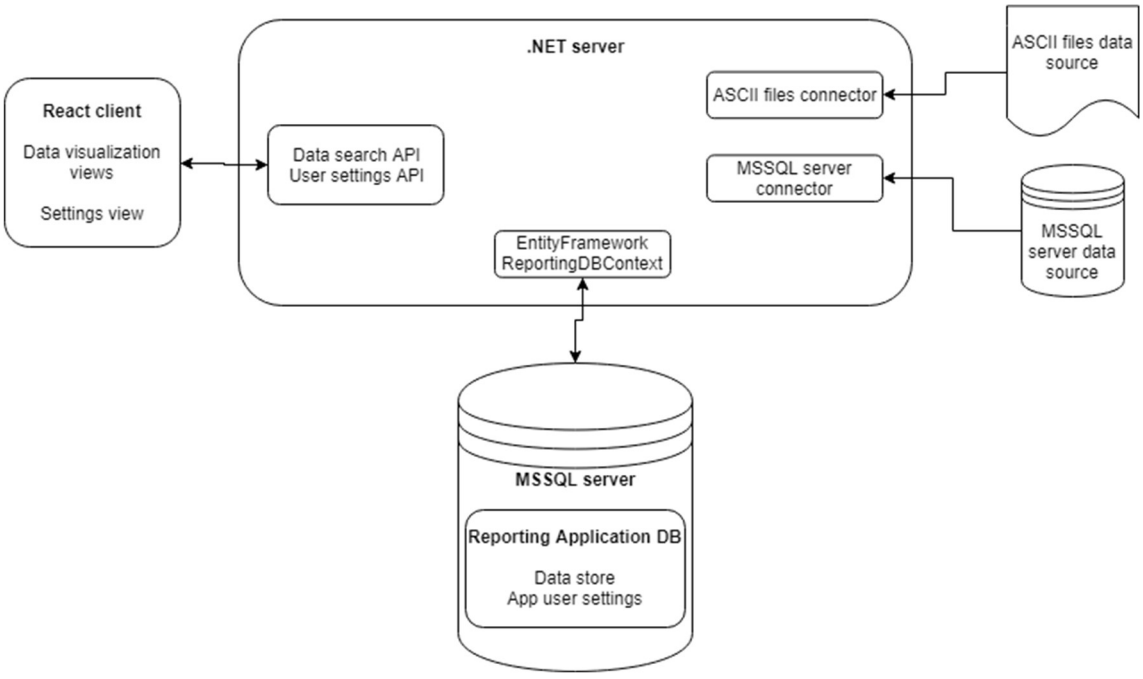

Figure 1. Reporting Application architecture

The details behind the implementation of each component of the architecture presented in Figure 1 are presented in the following sections. The next section explains the details behind ASCII files and MSSQL server data source connectors.

5.1 Data source layer and data source connectors

Reporting Application provides two data source connectors: an MSSQL server data source connector and an ASCII files data source connector. Both connectors are implemented by using the C# programming language and .NET framework. This section describes data source layer interfaces and classes implemented for this project from the group up; the usage of classes provided by the .NET framework and other external libraries is mentioned explicitly.

From the programming point of view all data source connectors implement the same interface IDataSourceConnector which provides methods for initial data import and scheduled data import, methods for error handling and a set of additional utility methods. IDataSourceConnector interface is implemented by LocalDataSourceConnector class. LocalDataSourceConnector instances are controlled by DataSourceConnectionManager class which resolves necessary data source connectors from the user-defined configuration. DataSourceConnectionManager is exposed to a REST API controller which makes it possible to start initial and scheduled data import operations by means of REST API. Figure 2 provides an overview of the relationships between interfaces and classes used in the implementation of the data source layer.

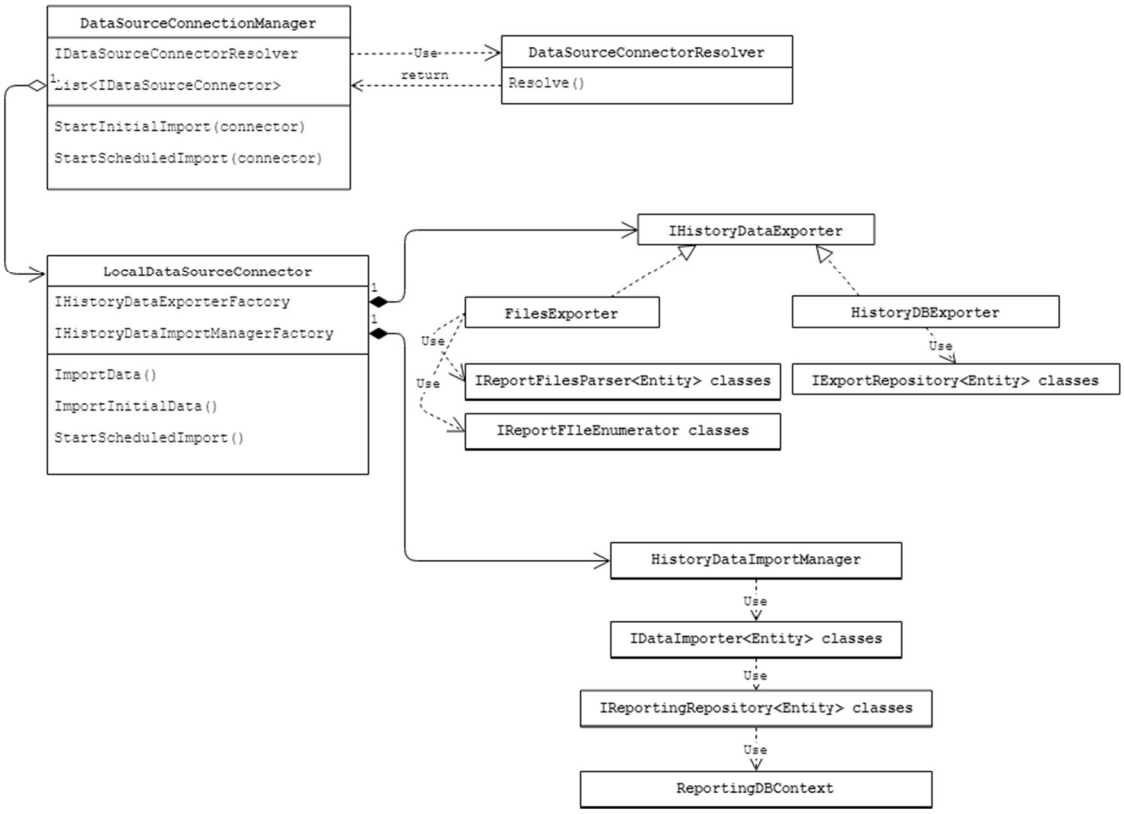

Figure 2. Relationships between the interfaces and classes used in the implementation of the data source layer

Only one data source connector class exists in the current implementation; however, composition is used to define the functionality of this class. Various classes implementing the IHistoryDataExporter interface can be provided to the LocalDataSource connector class. Provided implementation defines from which type of data source data will be exported. The classes, implementing the IHistoryDataExporter interface, are provided by means of dependency injection. Dependency injection in this project is implemented by using the Windsor inversion of control (IoC) container, which offers DI functionality by means of constructor injection. All dependencies shown in Figure 2 are handled by the Windsor IoC container.

Implementation details of MSSQL server history data exporter and ASCII files history data exporter are explained in the following two sections. Section 5.1.3 provides details on the implementation of history data importer. Section 5.1.4 explains how the application interacts with data source connectors.

5.1.1 MSSQL server history data exporter

MSSQL server history data exporter reads data from the MSSQL server HistoryDB database by using raw SQL queries. Use of raw SQL queries was chosen over the use of EntityFramework, as different versions of HistoryDB exist and the former method proved to be more flexible and efficient at handling differences between those versions.

The main data export logic resides in the HistoryDBExporter class, which implements the IHistoryDataExporter interface. The repository pattern is used to handle the connection to the database and read data from various database tables. HistoryDBExporter relies on a set of repository classes. Each repository class extends abstract class RawSqlExportRepository which implements IExportRespository interface. The IExportRepository interface provides methods to retrieve either all existing data at once or only data that belongs to a time range specified.

Repository classes use SqlConnection class provided by the .NET framework and a set of helper methods to perform raw SQL queries. The helper methods facilitate the writing of SQL queries and provide a possibility to apply various search filters and to check if database tables and columns specified exist.

5.1.2 ASCII files history data exporter

ASCII files history data exporter retrieves data from ASCII text files and from XML files. The implementation logic of the exporter resides in the FilesExporter class, which implements IHistoryDataExporter interface. The FilesExporter class relies on a set of file parser classes. Each file parser class implements generic interface IReportFilesParser, which provides methods to read either all data from all files of a specific type or only data for a time range specified.

ASCII file parsers use custom logic for reading data from the files, as different types of ASCII files have different formatting and a different set of character separators. XML file parser uses XmlSerializer class provided by the .NET framework to read data from the files. Both XML and ASCII file parsers rely on a set of file enumerator classes to correctly handle different patterns used in the names of the files. All file enumerator classes implement the IReportFileEnumetor interface. IReportFileEnumerator provides methods to get the names of either all files of a specific type or only of the files that belong to a time range specified.

The use of IHistoryDataExporter interface allows to hide the implementation details and to use dependency injection in an efficient manner. Moreover, this approach allows creating an API that provides data in a consistent manner, always using the same data models, no matter what the actual data source is. This approach also makes possible to entirely separate data export logic from data import logic. The details of the data import functionality implementation are described in the next section.

5.1.3 History data import manager

History data import functionality is implemented by the HistoryDataImportManager class. The HistoryDataImportManager class handles enhancement and transformation of the data provided by the history data exporter classes and writes the consolidated data to Reporting Application DB.

The classes extending the IHistoryDataImporter interface are responsible for enhanced and transformation of the entities exported by the history data exporter classes. The IHistoryDataImporter interface is a generic interface and each implementation of this interface handles one data model. History data importer classes calculate additional fields for the exported entities and derive new entities based on the exported data. Performing such calculations before writing the data to the database makes it possible to avoid additional reoccurring calculations while retrieving the data from datastore.

Repository pattern is used for implementation of interaction with the MSSQL database. This approach makes it possible to hide the database related implementation details and to swap the database engine used by the datastore with ease if necessary. Classes implementing IReportingRepository interface are used for implementing CRUD operations. The IReportingReporting repository interface is a generic interface and each class, implementing this interface, handles one data model. Entity Framework code-first approach is used for ORM implementation and the ReportingDBContext class, that extends the DBContext class provided by the Entity Framework, handles the ORM concerns.

5.1.4 Data source connection manager

The data source connectors are managed by the DataSourceConnectionManager class. Upon initialization, this class reads user settings from the database and creates necessary data source connector instances based on those user settings. An instance of DataSourceConnectionManager is created on application startup and starts scheduled data

import for the connectors if necessary. A DataSourceConnectionManager instance is also exposed via REST API to make it possible to trigger initial and scheduled data import from the client side.

The following example demonstrates the operation of the data source layer. The user sets up an MSSQL data source by using the client GUI. The client calls the REST API provided by Reporting Application. Reporting Application saves the data source settings provided by the user and uses DataSourceConnectionManager to create an instance of LocalDataSourceConnector, using an instance of HistoryDBExporter class. This operation is illustrated in Figure 3.

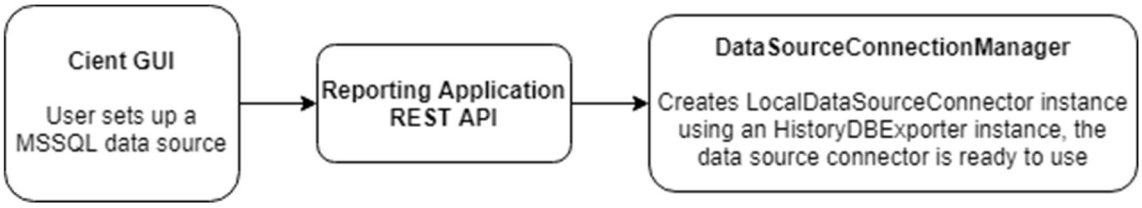

Figure 3. Interaction between Reporting Application components during the creation of an MSSQL data source connector

The user starts an initial data import from the added data source by using the client GUI. The client calls the Reporting Application REST API which uses an instance of the DataSourceConnectionManager class to handle the start of the initial data import operation. Then DataSourceConnectionManager instance uses the corresponding LocalDataSourceConnector instance to perform the initial data import operation. The LocalDataSourceConnector instance handles export and import for each data model available from the HistoryDB separately. It uses an instance of IHistoryDataExporter corresponding to a specific data model to retrieve all entities of this type from HistoryDB, then it uses an instance of IDataImporter corresponding to that entity type to perform the data enhancement, and finally it uses an IReportingRepository instance corresponding to that entity type to write the enhanced entities to the Reporting Application dedicated database. This process is demonstrated in Figure 4.

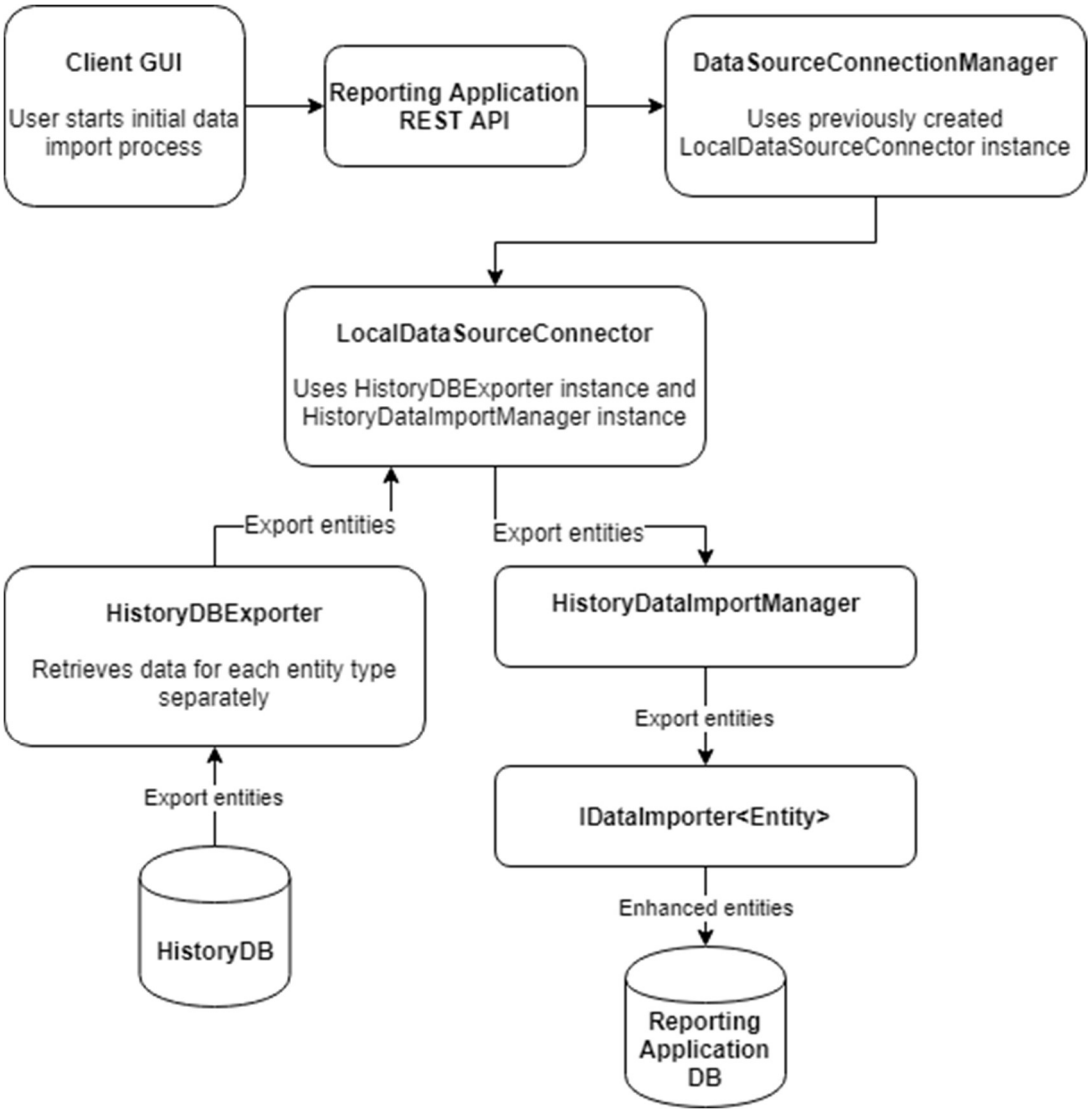

Figure 4. Initial data import operation, where the data is being imported from an MSSQL data source

The data source connection manager handles scheduled data import operations and data import operations for ASCII files data source in a way, similar to the described above. The next section provides details concerning aggregation operations performed on the already enhanced and transformed data retrieved from Reporting Application DB.

5.2 Data aggregation logic

Reporting Application provides a search API heavily influenced by Elasticsearch API. The Reporting Application search API provides a set of flexible aggregation mechanics implemented from the ground up. Though the style of the API, in terms of aggregation functionality, is very similar to the API provided by Elasticsearch, a decision was made to proceed with a custom implementation. There are two reasons behind this decision: first - the need to keep the production environment as simple as possible, second - the need to implement custom metrics aggregation specific to Reporting Application. The following sections explain how data is retrieved from the Reporting Application datastore, processed by the layer of aggregators and then provided to the REST API controllers. The overview of the process is demonstrated in Figure 5.

Figure 5. Flow of data from Reporting Application DB, through the aggregation layer, to client

Figure 5 demonstrates the general idea of how data flows from Reporting Application datastore, is processed by aggregators and then is presented to the client by means of search API. The search API usage and mechanics are explained in more detail in section 5.3. Section 5.2 focuses on the implementation details of the aggregation layer.

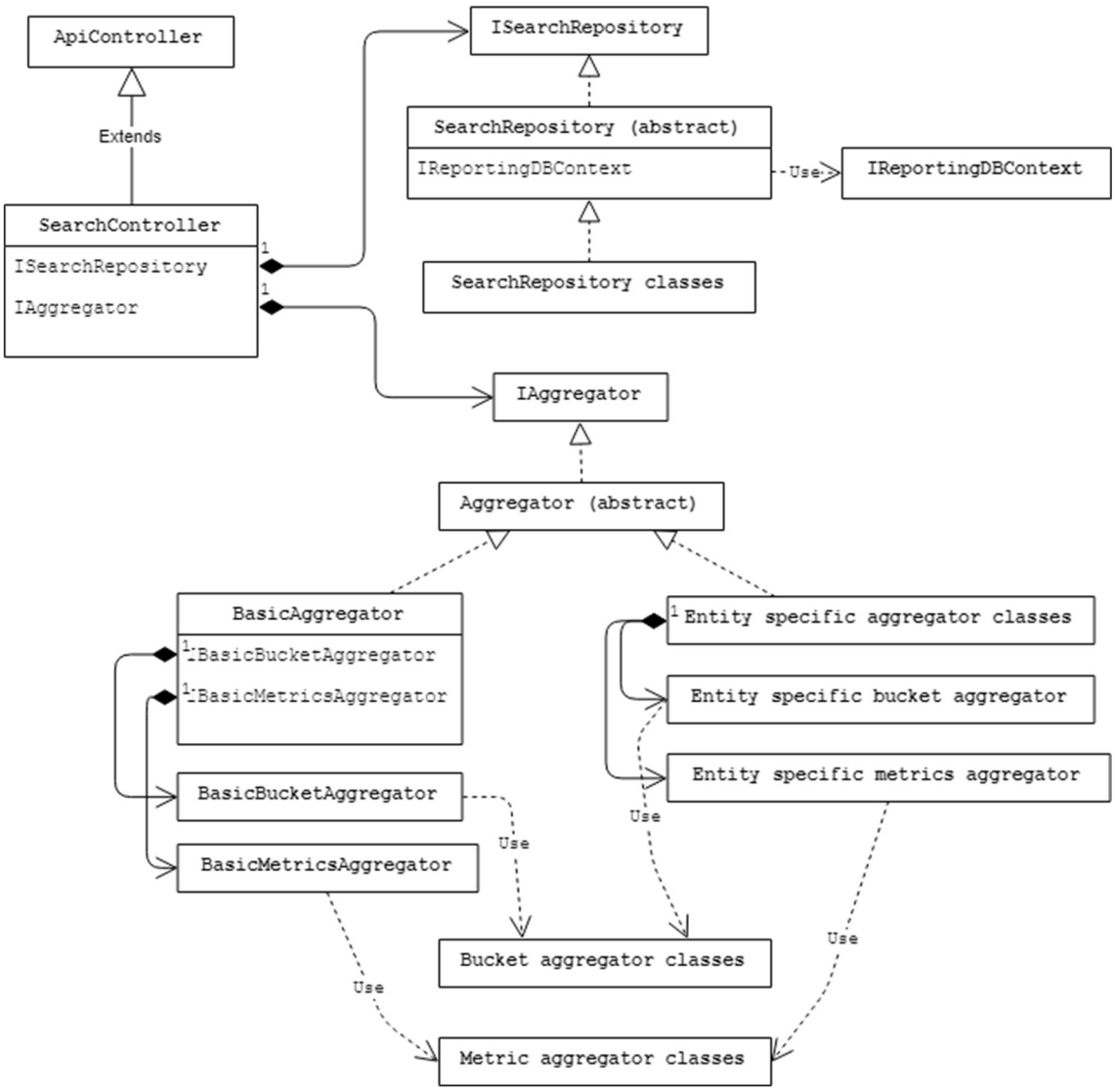

Figure 6. Relationships between the classes used in the implementation of the data aggregation logic

Figure 6 demonstrates the general idea behind the relationships between the classes used in the implementation of aggregation logic. Reporting Application uses Entity Framework and the same ReportingDBContext, mentioned in section 5.1.3, for ORM purposes and for retrieving data from its own MSSQL database. The Entity Framework context is wrapped in a set of repository classes, like in the case of data source layer implementation. However, to achieve better code decoupling and stronger separation of concerns a different set of repository classes, which implement the ISearchRepository interface, is used. Each class implementing the ISearchRepository interface handles one entity type. These repository classes are used by search API controllers. Each search API controller extends the generic SearchController classes, which extends the ApiController class provided by the .NET WebApi framework. Search controller classes handle

the HTTP requests, retrieve entities from search repository classes and pass the retrieved entities and the search request to the aggregator classes.

5.2.1 Data aggregation concepts

In a way similar to Elasticsearch, Reporting Application provides two types of aggregation: bucket aggregation and metrics aggregation. The bucket aggregation makes it possible to filter and group data entities by different criteria in a recursive manner. This type of aggregation is discussed in more detail in the following section. The metrics aggregation makes it possible to calculate and derive new values based on the values that a data entity or a group of data entities already hold.

The aggregation layer is composed of a set of classes that perform different tasks. The aggregation entry point is provided by classes implementing a generic IAggregator<Entity> interface. Each class, implementing the IAggregator interface, handles aggregation for only one entity type. These classes read the search request in a recursive manner and construct a chain of aggregators defined by the request. The data retrieved from Reporting Application datastore is being processed through this chain of aggregators and is provided by means of the search API.

Beside the IAggregator classes, which control the flow of the aggregation, a range of classes is implemented that handle the specific steps of the aggregation chain. These aggregators can be divided into two groups: bucket aggregators, such as time range, time histogram, and term aggregators and metrics aggregators, such as sum, average value, and document aggregators.

5.2.2 Bucket aggregators

The main purpose of bucket aggregators is to provide a flexible way to filter and group entities by a range of criteria. Reporting Application relies on numerous classes providing different aggregation functionality, such as time range aggregation, shift time range aggregation, time histogram aggregation, shift time histogram aggregation, term aggregation, term filter aggregation, and range aggregation. Each bucket aggregation can have an unlimited number of nested bucket sub-aggregations, this recursive mechanic ensures the flexibility of the search API.

The following example demonstrates an aggregation involving multiple bucket aggregation types. The BendingPartProduction model is used in the example and it has properties as shown in Figure 7. The aggregation example itself is illustrated in Figure 8. The goal is to retrieve to BendingPartProduction entities that belong to time range 01.01.2018 – 01.02.2018, filter out the entities that have PartsDone property with a value of 0, and finally group the entities by the value of the Name property.

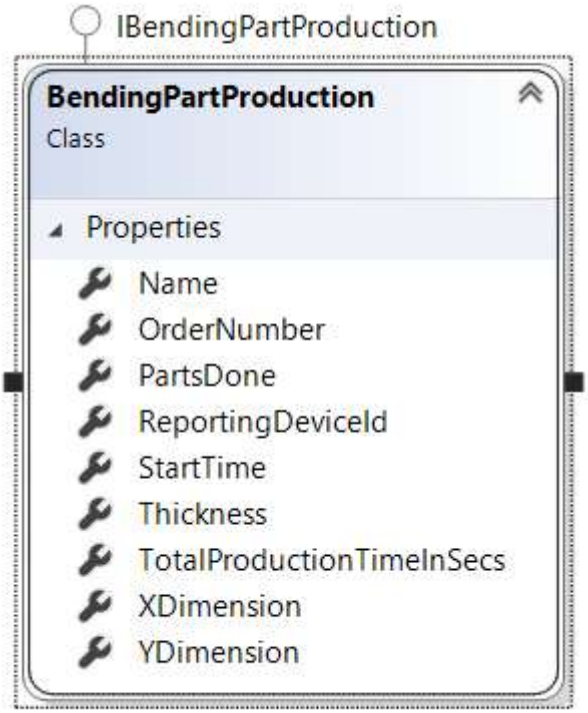

Figure 7. Properties of the BendingPartProduction model

Search API is discussed in more detail in section 5.3.2, this example focuses on demonstrating the interaction between the bucket aggregator classes. Assume a correctly constructed search API request and lazy loaded BendingPartProduction entities were passed to an instance of the TimeEntityAggregator class, one of the entity-specific aggregator classes depicted in Figure 6. The TimeEntityAggregator class instance starts to read the search request recursively, it reads the root of the request first and determines that time range aggregation should be performed. It passes the entities to the entity-specific bucket aggregator class TimeEntityBucketAggregator, which passes the entities to BendingPartProductionTimeRangeAggregator class instance. This instance filters the BendingPartProduction entities and returns only the entities that belong to the previously specified time range 01.01.2018 – 01.02.2018.

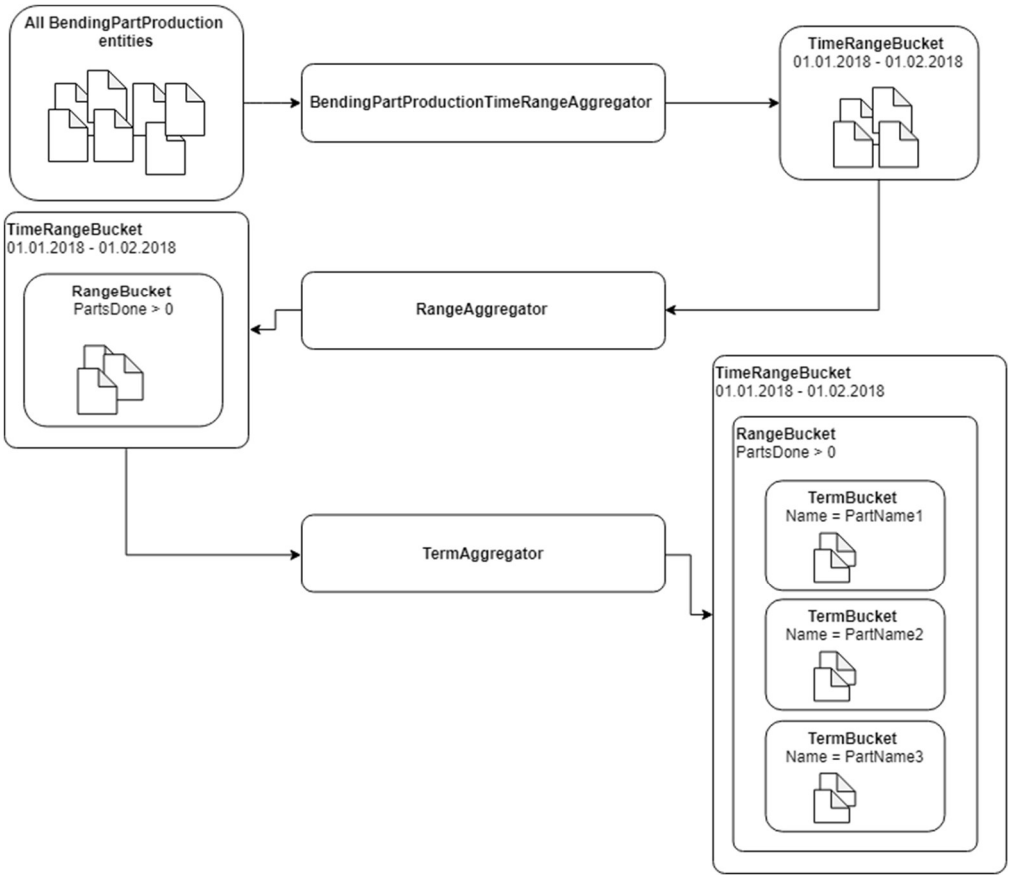

Figure 8. Aggregation process involving multiple nested bucket aggregations

Next, the control is returned to the TimeEntityAggregator instance, which continues to read the request recursively and determines that a range aggregation should be performed. It passes the filtered BendingPartProduction entities to the TimeEntityBucketAggregator instance again, which passes the entities to a RangeAggregator instance. Which filters out the entities that have PartsDone property with a value less than 1.

The control again returns to the TimeEntityAggregator instance, which continues to read the request recursively, passes the filtered entities to the TimeEntityBucketAggregator instance, which passes the entities to a TermAggregator instance. This instance groups the entities by the value of the Name property.

The control returns to the TimeEntityAggregator instance, which determines that there are no further aggregations specified in the search request and returns the result containing performed nested bucket aggregations. The final aggregation result can be seen in Figure 8. Besides the final aggregation result, entities grouped by the value of Name

property and filtered by time range and value range, it contains also all the intermediate aggregation results, which can be used separately if necessary.

5.2.3 Metrics aggregators

The main purpose of metrics aggregator is to calculate and derive values based on the values of the properties of the entities being aggregated. A metrics aggregation can be a sub-aggregation of any bucket aggregation, however, it itself cannot contain any sub-aggregations. Reporting Application provides a range of metrics aggregations, which includes more general aggregations, such as document, sum, average, minimum and maximum aggregations that can be performed on entities of all types, as well as aggregations, such as overall equipment effectiveness, mean time between failures and usability rate that can be performed only on the entities of a certain type.

The example concerning bucket aggregations from section 5.2.2 can be extended to demonstrate how the application logic handles the metrics aggregations. In the example from section 5.2.2, the final BendingPartProduction aggregation result contained a root time range aggregation and two nested sub-aggregations: a range and a term bucket aggregation. In other words, BendingPartProduction entities filtered by the time range specified, filtered by the value of the PartsDone property and grouped by the value of the Name property. Assume the search request was modified to include an instruction to perform an average value metrics aggregation on the PartsDone property of BendingPartProduction entities, as a sub aggregation of the last nested term aggregation. In this case, when the TermAggregator instance finishes its job the control returns to the TimeEntityAggregator instance, which continues to read the search request and determines that a metrics average value aggregation should be performed next.

The TimeEntityAggregator instance passes each term aggregation bucket, containing BendingPartProduction entities, to an instance of AvgAggregator. The average value for the PartsDone property is being calculated separately for each term bucket by a separate AvgAggregator instance. The aggregation process and its result are shown in Figure 9.

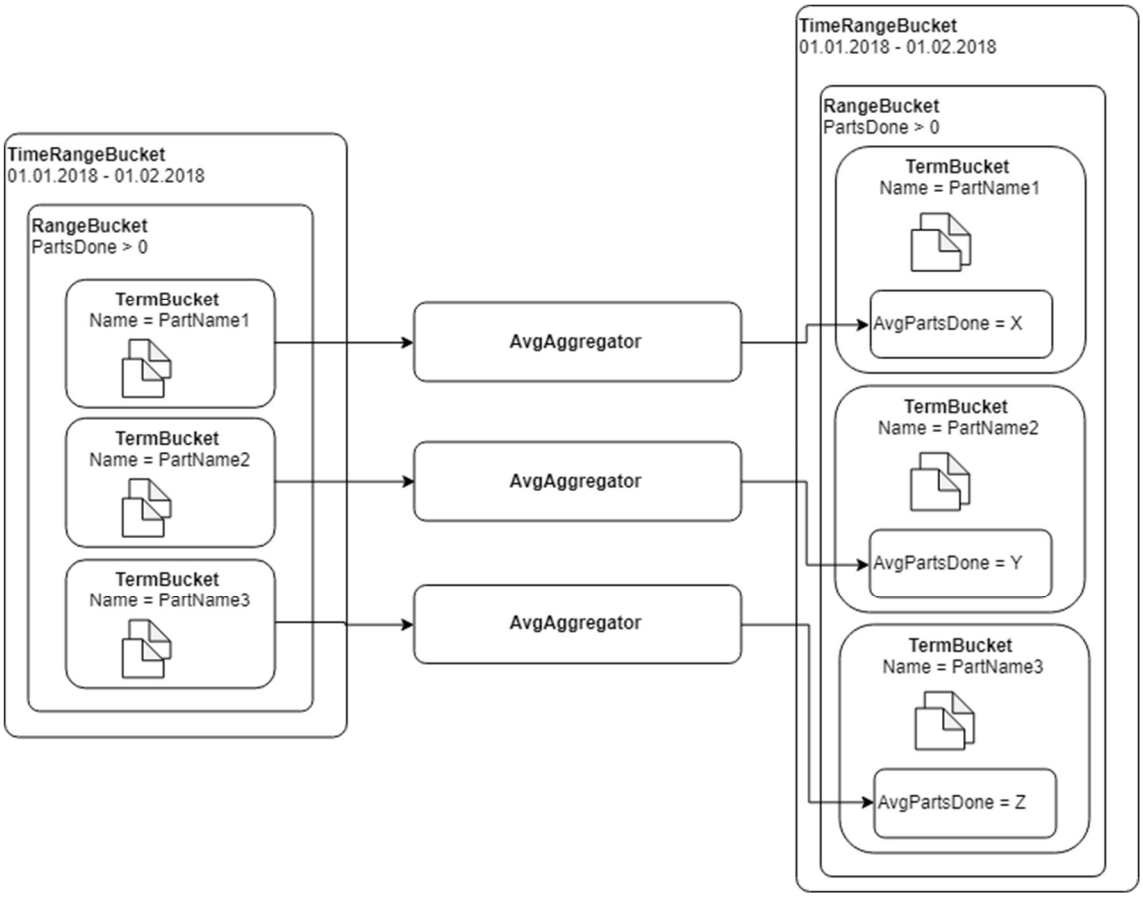

Figure 9. Metrics average value aggregation process and its final result

Metrics aggregation can be a sub-aggregation of a bucket aggregation at any level of nesting. The previous example could be modified so, that another average value aggregation would be set as a sub aggregation of the range bucket aggregation. In this case, the average value would be calculated both for all entities filtered by time range and by the value of the PartsDone property and for the entities in each term bucket individually. These aggregation mechanics and the use of the search API is explained in more detail in sections 5.3.2 and 5.3.3.

5.3 REST API

Reporting Application provides two types of REST API. The first type is a more traditional REST API, providing CRUD functionality for handling tasks related to user configuration, such as managing of data sources, various user settings, and user-defined dashboards. At the moment, Reporting Application does not have a user management system and authentication, however, plans exist to implement such functionality in the future. The second type of API is the data search API, it provides ways to query and aggregate data

in an Elasticsearch-like manner. This section focuses on the explanation of the usage of the data search API.

5.3.1 Web API controllers

The REST API implementation is based on the functionality provided by the .NET Web API framework. All of the REST API controllers used in the implementation of Reporting Application extend the ApiController class provided by the .NET Web API framework. Reporting Application uses two types of REST controllers. The first type of controllers backs up the more traditional API related to user configuration management. These controllers extend the ApiController class directly and handle CRUD operations by processing GET, POST, PUT and DELETE HTTP requests to predefined routes.

The second type of controllers is the data search API controllers. These controllers extend the SearchController class, which extends the ApiController class provided by the .NET Web API framework. These controllers handle only POST HTTP requests with a JSON search request that has a specific format. The JSON parsing and serialization concerns are handled by the Newtonsoft Json.NET framework. The search request format is discussed in more detail in the following section.

5.3.2 Data search API concepts

Data search REST API provides functionality to query and aggregate the data from Reporting Application datastore. Data search REST API implementation is provided by the search controller classes which use data aggregator classes. A search endpoint usually has the following form ”/api/machines/{machine-id}/{entity-index}/search”. The machine ID value is generated when the user configures a data source, this value determines to which machine the imported data to be searched belongs. Entity index determines the type of data to be searched, the example provided in section 5.2.2 used the BendingPartProduction data model, the entity index for this model is ”bending-part-production”. The example provided in this section explains how to use the data search API to perform the aggregation operation explained in section 5.2.2.

The search request has a form shown in Listing 1. The JSON property ”aggs” marks the start of the collection of aggregation definitions. The object assigned to the property ”aggs” is a collection of aggregation definitions. Each property of the collection corresponds to the name of the aggregation definition. In the search request example from

Listing 1, the root definition collection has only one aggregation definition with the name ”myTimeRangeAgg”. The name of the aggregation can be chosen by the developer freely. The object assigned to the property ”myTimeRangeAgg” is the aggregation definition.

The aggregation definition object has two properties ”timeRange” and ”aggs”. The property ”timeRange” is a mandatory aggregation type property. The object assigned to the ”timeRange” property is an aggregation parameters object and it has a strict specific format. In this case, the aggregation parameters object tells the back-end to perform a time range bucket aggregation and to filter the BendingPartProduction entities by time range 01.03.2018 - 01.04.2018. The ”aggs” property defines the start of the sub-aggregation definition collection. The sub-aggregation definition collection object follows the same syntax rules as the root aggregation definition collection.

In the example shown in Listing 1, a range bucket aggregation with name ”myRangeAgg” is defined as a sub-aggregation of ”myTimeRangeAgg” bucket time range aggregation. This definition tells the back-end to perform a range aggregation and to filter the BendingPartProduction entities with the value of the PartsDone property greater than 0. The ”myRangeAgg” aggregation has its own sub-aggregation with name ”myTermAgg” of term aggregation type. The definition of this aggregation means that the BendingPartProduction entities should be grouped by the value of the property Name.

```
Route: /api/machines/2/bending-part-production/search

{
      "aggs": {
            "myTimeRangeAgg": {
                  "timeRange": {
                        "ranges": [{"from": "2018-03-01", "to": "2018-04-01"}]
                  },
                  "aggs": {
                        "myRangeAgg": {
                              "range": {
                                    "field": "partsDone",
                                    "ranges": [{"from": 0}]
                              },
                              "aggs": {
                                    "myTermAgg": {
                                          "term": {
                                                "field": "name"
                                          }
                                    }
                              }
                        }
                  }
            }
      }
}
```

Listing 1. Example of data search API JSON request.

The search request shown in Listing 1 provides a search result demonstrated in Listing 2. The structure of the response repeats the structure of the search request. The response JSON object contains the results of the “myTimeRangeAgg” root aggregation and two sub-aggregations “myRangeAgg” and “myTermAgg”. Each bucket aggregation result contains a “buckets” property. An array of bucket objects is assigned to the “bucket” properties. Each bucket object contains a “count” property and an additional set of properties, which depend on the aggregation type performed. The “count” property tells how many of the aggregated entities belong to this bucket.

```
{
    "aggs": {
        "myTimeRangeAgg": {
            "buckets": [
                {
                    "from": "2018-03-01T00:00:00",
                    "to": "2018-04-01T00:00:00",
                    "count": 569,
                    "myRangeAgg": {
                        "buckets": [
                            {
                                "field": "partsDone",
                                "from": 0,
                                "to": null,
                                "count": 569,
                                "myTermAgg": {
                                    "buckets": [
                                        {
                                            "count": 38,
                                            "key": "Part 2"
                                        },
                                        {
                                            "count": 37,
                                            "key": "Part 1"
                                        },
                                        {
                                            "count": 35,
                                            "key": "Part 4"
                                        },
                                        ...
                                    ]
                                }
                            }
                        ]
                    }
                }
            ]
        }
    }
}
```

Listing 2. Search response to request displayed in Listing 1.

In the example from Listing 2, the “myTimeRangeAgg” aggregation contains only one bucket which contains 569 BendingPartProduction entities. The bucket also has properties specific only to the time range aggregation type: a “from” property and a “to” property which hold respective timestamp values of the time range specified in the search request.

The “myRangeAgg” sub-aggregation contains only one bucket with 569 BendingPartProduction entities. This bucket has three additional properties specific to the range aggregation type. The “from” and “to” properties hold the range values specified in the definition of the range aggregation in the search request. The “field” property value is the name of the data model property which was used to filter the entities by value range, in this case, this the PartsDone property of the BendingPartProduction model.

The “myTermAgg” sub-aggregation contains numerous buckets, the example from Listing 2 shows only the first three buckets. Each of those buckets has a “count” property and a “key” property. The “key” property holds the value of the PartName property of the BendingPartProduction model by which the entities were grouped. The example above might not be very practical, as it provides only count aggregation, but it clearly demonstrates the concepts of bucket aggregation in action.

The following example explains how bucket aggregations can be combined with metrics aggregations. The example displayed in Listing 3 extends the bucket aggregation example from Listing 1 with an average value metrics aggregation.

```
Route: /api/machines/2/bending-part-production/search

{
      "aggs": {
            "myTimeRangeAgg": {
                  "timeRange": {
                        "ranges": [{"from": "2018-03-01", "to": "2018-04-01"}]
                  },
                  "aggs": {
                        "myRangeAgg": {
                              "range": {
                                    "field": "partsDone",
                                    "ranges": [{"from": 0}]
                              },
                              "aggs": {
                                    "myTermAgg": {
                                          "term": {
                                                "field": "name"
                                          },
                                          "aggs": {
                                                "myAvgAgg" :{
                                                      "avg": {"field": "partsDone"}
                                                }
                                          }
                                    }
```

```
                        }
                    }
                }
            }
        }
}
```

Listing 3. Example of a search request containing definitions of three bucket aggregations and a definition of an average value metrics aggregation with the name “myAvgAgg”.

The search request displayed in Listing 4 shows the first three term buckets of the “myTergAgg” aggregation. The parent aggregation results are omitted for the sake of brevity. Each term bucket now contains a “myAvgAgg” property which holds an object with a property “value”. The property “value” contains the value of the average aggregation result performed on the ParsDone property of the BendingPartProduction entities that belong to this term bucket.

```
. . .
      "myTermAgg": {
         "buckets": [
             {
                 "count": 38,
                 "key": "Part 2",
                 "myAvgAgg": {
                     "value": 272.15789473684208
                 }
             },
             {
                 "count": 37,
                 "key": "Part 1",
                 "myAvgAgg": {
                     "value": 297.43243243243245
                 }
             },
             {
                 "count": 35,
                 "key": "Part 4",
                 "myAvgAgg": {
                     "value": 290.77142857142854
                 }
             },
             . . .
         ]
      }
. . .
```

Listing 4. Part of the response to search request shown in Listing 3. Parent aggregation results are omitted for brevity purposes.

The examples provided in this section demonstrate the flexibility of the Reporting Application data search API, however, this flexibility comes at a cost. It can be seen from the examples provided above that both the search requests and the responses are quite verbose and include numerous nested objects. Section 5.4 explains some of the details behind the React – Redux client implementation, including the approach that was chosen at the client side to work with the data search API.

5.4 Presentation layer

The core libraries used in the implementation of the presentation layer are React, Redux and C3.js. Flow static type checker is used to increase the stability of the codebase. The client application is bootstrapped by using react-scripts Node.js package, which provides a development Node.js server and a preconfigured webpack bundler configuration. The react-scripts package handles the creation of the client production build by using a pre-defined webpack configuration. In production, the client application is hosted by the Reporting Application back-end.

The client application provides GUI views of three types: pre-defined dashboard views, user-defined dashboard views and settings views. The largest part of the codebase is related to the implementation of the dashboard widgets containing various data visualizations and fetching data for those visualizations from the back-end by using the data search API.

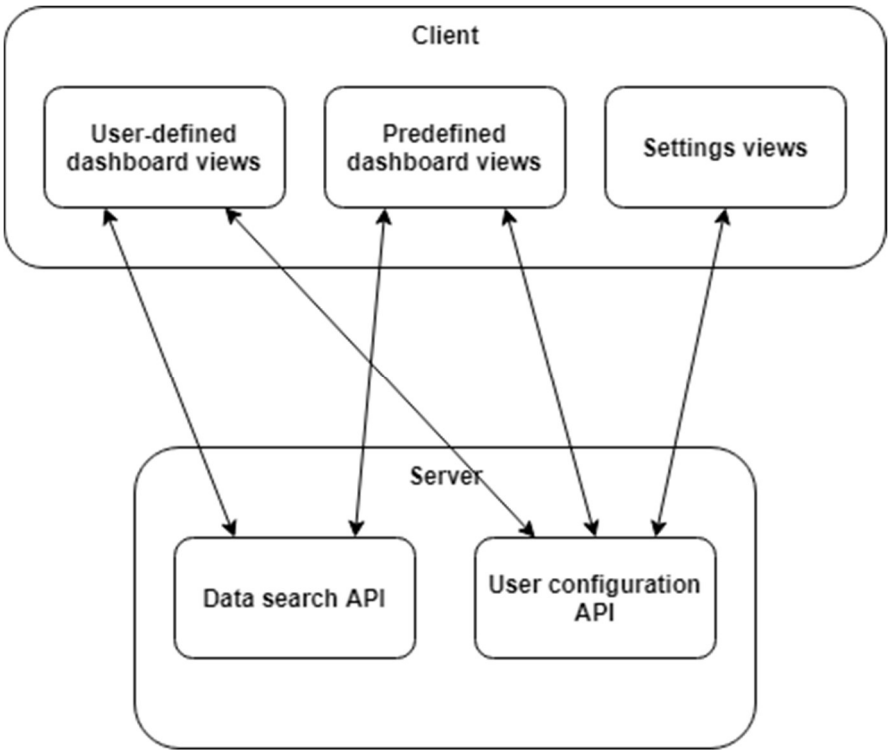

Figure 10. General idea behind the client-server interaction.

The diagram in Figure 10 illustrates the general idea behind client-server interaction. The settings views interact only with the user configuration API, however, the dashboard views need both the data provided by the user configuration API to define correct search requests and the data provided by the data search API to back up the data visualization components. The client application needs to handle a large amount of data; thus, a state management library is a necessity. Redux was chosen for this purpose, as it is a part of a well-established React – Redux client application stack.

5.4.1 Dashboard widgets and data visualizations

A range of custom dashboard widget is implemented for Reporting Application client. Each widget uses a higher order React component handling data management concerns and a data visualization component handling rendering tasks. Reporting Application uses four approaches for rendering data visualizations. Various charts are rendered by using C3.js library, list visualizations are rendered by using React Table library, timeline visualizations are rendered by using Vis.js library and more simple visualizations, such as count and trend visualization, are rendered by using basic HTML elements.

Each widget React component has its own Redux state and a set of action-creator functions that handle data fetch requests and dispatch the fetched data to the Redux store. The action-creator functions use a set of client functions, which handle the creation of a data search request body, and a set of parser functions, which transform a complex data search response body into a flatter data structure that can be consumed by a visualization component.

Each widget component requires an object property with the name “searchParams”. The “searchParams” object carries a “dataType” property that defines which client function should be used for constructing the data search request. The “searchParams” object also has a set of additional data search parameter properties which are used by the client function for setting the values of the aggregation definition objects of the data search request. The “datatype” property also defines which parser function should be used to process the data search response. Figure 11 illustrates the data flow for a single widget component.

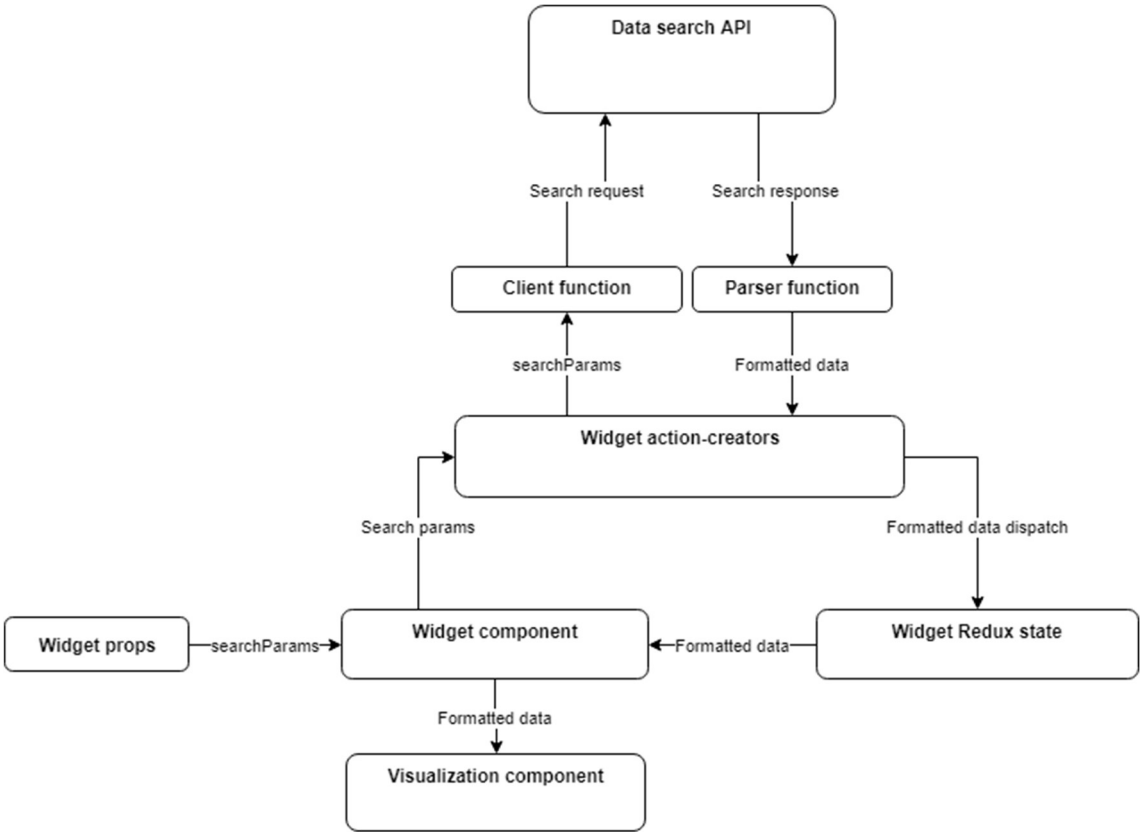

Figure 11. Data flow for a single widget component.

The “searchParams” objects and client functions are logically divided into four types, depending on the type of the root bucket aggregations. The types are as following: time range, time histogram, term filter and trend (a special type that has a time range root aggregation with two separate time ranges). The parser functions are also logically divided into several types depending on the type of visualization they are meant to parse data for. Examples of these types could be time histogram, ratio, document, and trend. The following section provides examples of dataflow for two of the widget implementations used in Reporting Application.

5.4.2 Widget data flow examples

The first example explains in detail the dataflow for the widget “Last 5 parts” from Reporting Application bending part analysis view. The bending part analysis view is one of the predefined dashboard views implemented for Reporting Application. This view displays a number of widgets related to bending part production for a time range specified. The time range for the view can be set by using GUI controls. The “Last 5 parts” widget displays the number of parts done by 5 most recent bending part production programs

that belong to the time range specified by using a pie chart visualization. The “Last 5 parts” widget is shown in Figure 12.. This time range value is passed to the RatioWidget component via React component props and then is a passed as one of the parameters to the getLastBendingParts function. The function uses a number of helper functions to create a data search request body that can be seen in Listing 7.

```
Route: /api/machines/2/bending-part-production/search

{
  "aggs": {
    "timeRangeAgg": {
      "timeRange": {
        "ranges": [{
          "from": "2018-02-01T00:00:00",
          "to": "2018-02-28T23:59:59",
          "datesToExclude": []
        }]
      }, "aggs": {
        "documentAgg": {
          "document": {
            "size": 5,
            "order": { "startTime": "desc" }
          }
        }
      }
    }
  }
}
```

Listing 5. Search request body created for the widget “Last 5 parts”.

The search request body shown in Listing 7 has a time range bucket aggregation definition with the name “timeRangeAgg” in its root and a nested document metrics aggregation definition with the name “documentAgg”. The “documentAgg” definition contains two parameters: size and order. The order parameter means that the BendingPartProduction entities (see Figure 7 for the data model details) should be ordered by the value of the StartTime property in descending order. The size parameter means that only the five first BendingPartProduction should be included in the response. The search response body can be seen in Listing 8.

```
{
    "aggs": {
        "timeRangeAgg": {
            "buckets": [
                {
                    "from": "2018-02-01T00:00:00",
                    "to": "2018-02-28T23:59:59",
                    "count": 37,
                    "documentAgg": {
                        "documents": [
                            {
                                "entityId": 38,
```

```
            "reportingDeviceId": 2,
            "startTime": "2018-02-28T23:57:48.93",
            "name": "Part 14",
            "orderNumber": "Order_0",
            "thickness": 1.7,
            "xDimension": 785,
            "yDimension": 939,
            "partsDone": 228,
            "totalProductionTimeInSecs": 256
        },
        {
            "entityId": 36,
            "reportingDeviceId": 2,
            "startTime": "2018-02-28T19:52:56.847",
            "name": "Part 12",
            "orderNumber": "LongOrder_000",
            "thickness": 1.3,
            "xDimension": 944,
            "yDimension": 631,
            "partsDone": 296,
            "totalProductionTimeInSecs": 821
        },
        {
            "entityId": 35,
            "reportingDeviceId": 2,
            "startTime": "2018-02-28T19:46:25.61",
            "name": "Part 5",
            "orderNumber": "Order_0",
            "thickness": 1.8,
            "xDimension": 943,
            "yDimension": 560,
            "partsDone": 64,
            "totalProductionTimeInSecs": 565
        },
        {
            "entityId": 34,
            "reportingDeviceId": 2,
            "startTime": "2018-02-28T19:32:10.237",
            "name": "Part 0",
            "orderNumber": "LongOrder_000",
            "thickness": 1.3,
            "xDimension": 669,
            "yDimension": 618,
            "partsDone": 101,
            "totalProductionTimeInSecs": 461
        },
        {
            "entityId": 33,
            "reportingDeviceId": 2,
            "startTime": "2018-02-28T19:21:43.347",
            "name": "Part 1",
            "orderNumber": "LongOrder_000",
            "thickness": 1.8,
            "xDimension": 693,
            "yDimension": 929,
            "partsDone": 380,
            "totalProductionTimeInSecs": 409
        }
    ]
. . .
```


Listing 6. Response to the search request for the “Last 5 parts” widget, displayed in Listing 7.

The search response displayed in Listing 8 is processed by the parseLastBendingParts function. The function flattens the search result structure and transforms the data into the format required by the RatioVisualization component. Then the formatted data is dispatched by the action-creator function to the Redux store. Figure 13 shows the state of the widget containing the formatted data that can be passed to the RatioVisualization component.

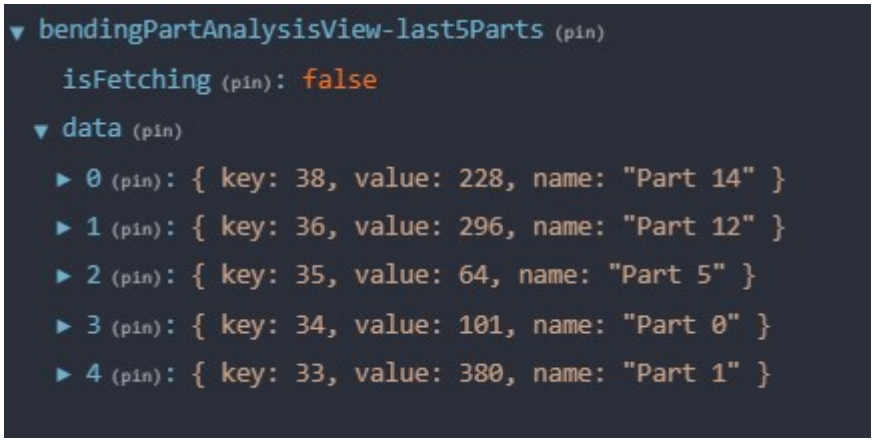

Figure 12. Screenshot of the Redux DevTools utility displaying the Redux state of the “Last 5 parts” widget after the search response was received and converted to the format required by the RatioVisualization component.

The second example explains the data flow for the “Part count” widget from the bending part analysis view. The widget, seen in Figure 14, displays a time series bar chart displaying how many parts with a given name were produced on each day of the time range specified. The time range used in the example is the same as in the previous example: 01.02.2018 – 28.02.2018. The “Part count” widget is implemented by using the HistogramWidget component which uses the HistogramVisualization component for rendering purposes. The HistogramVisualization component provides several rendering options such as bar, line and area charts and a list visualization. The “Part count” widget uses the stacked bar chart visualization option.

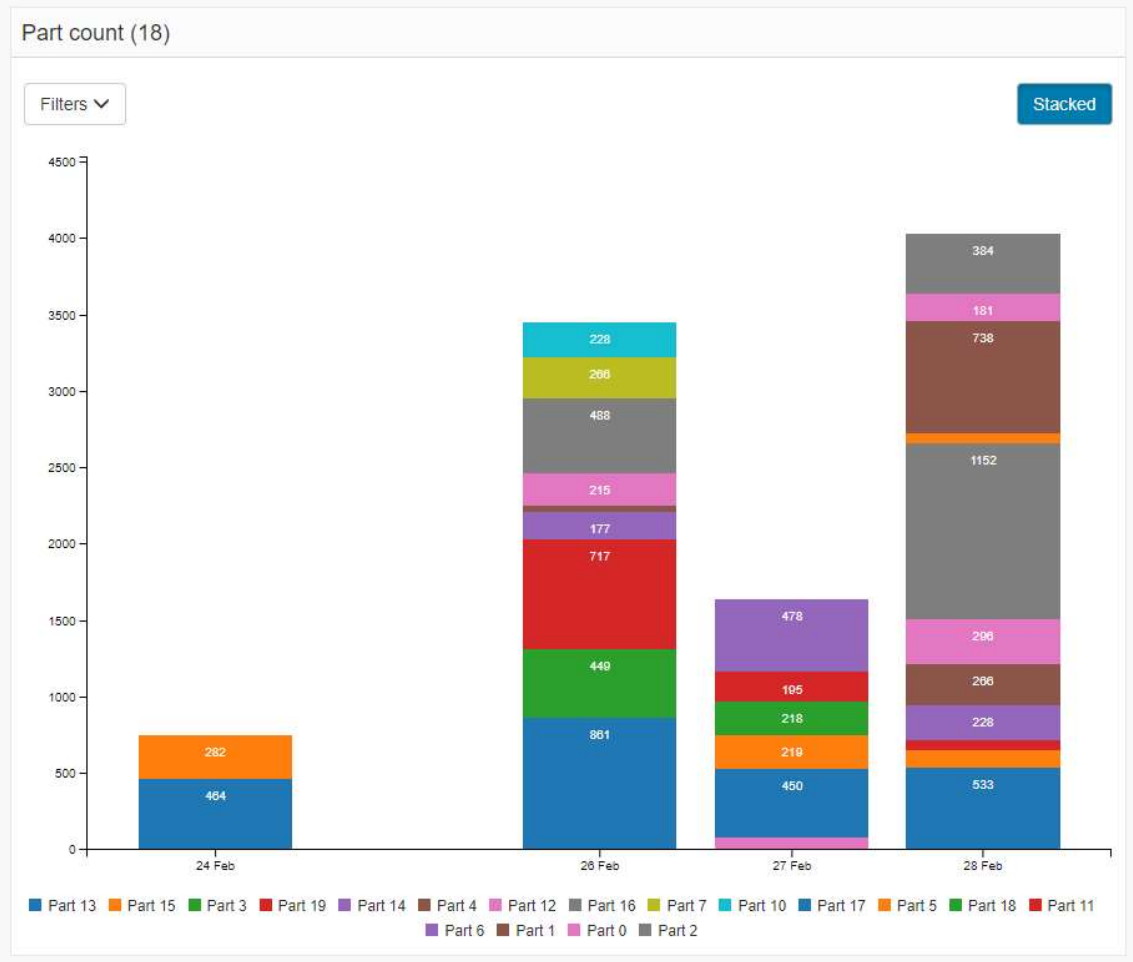

Figure 13. Screenshot of the “Part count” widget from the bending part analysis view of the Reporting Application client.

The getBendingPartsDoneGroupedByName function handles the creation of the search request for the “Part count” widget. The search request body produced by this function can be seen in Listing 9. The search request contains a time range bucket aggregation definition with the name “timeRangeAgg” in its root, two nested bucket aggregation definitions – a time histogram bucket aggregation definition with the name “timeHistogramAgg” and a term bucket aggregation definition with the name “termAgg”, and a nested sum metric aggregation definition with the name “sumAgg”. The “timeHistogramAgg” definition has default parameters, which means that the step of the time histogram aggregation is one day. The “termAgg” definition has the parameter “field” set to value “name”, which means that the BendingPartProduction entities inside the time histogram buckets will be grouped by the value of the Name property. The “sumAgg” definition’s parameter “field” is set to value “partsDone”; this will result in performing a sum aggregation on the property PartsDone for each term bucket of the “termAgg” aggregation.

```
Route: /api/machines/2/bending-part-production/search

{
  "aggs": {
    "timeRangeAgg": {
      "timeRange": {
        "ranges": [{
          "from": "2018-02-01T00:00:00",
          "to": "2018-02-28T23:59:59"
        }]
      },
      "aggs": {
        "timeHistogramAgg": {
          "timeHistogram": { "datesToExclude": [] },
          "aggs": {
            "termAgg": {
              "term": { "field": "name" },
              "aggs": {
                "sumAgg": {
                  "sum": { "field": "partsDone" }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 7. Data search request created for the “Parts done” widget.

The response to the data search request described above is shown in Listing 10. The response body is processed by the parseBendingPartsDoneGroupedByName parser function. This function transforms the response structure to the format required by the HistogramVisualization component. The formatted data, dispatched to the Redux store, can be seen in Figure 15.

```
{
  "aggs": {
    "timeRangeAgg": {
      "buckets": [{
        "from": "2018-02-01T00:00:00",
        "to": "2018-02-28T23:59:59",
        "count": 37,
        "timeHistogramAgg": {
          "buckets": [{
            "from": "2018-02-24T00:00:00",
            "to": "2018-02-25T00:00:00",
            "count": 2,
            "termAgg": {
              "buckets": [
                {
                  "count": 1,
                  "key": "Part 13",
                  "sumAgg": { "value": 464.0 }
                }, {
                  "count": 1,
                  "key": "Part 15",
                  "sumAgg": { "value": 282.0 }
```

```
            }]
        }
    },
. . .
```

Listing 8. Response to the search request for the “Part count” widget, displayed in Listing 9. Only the first time histogram bucket is shown, the rest is omitted for the sake of brevity.

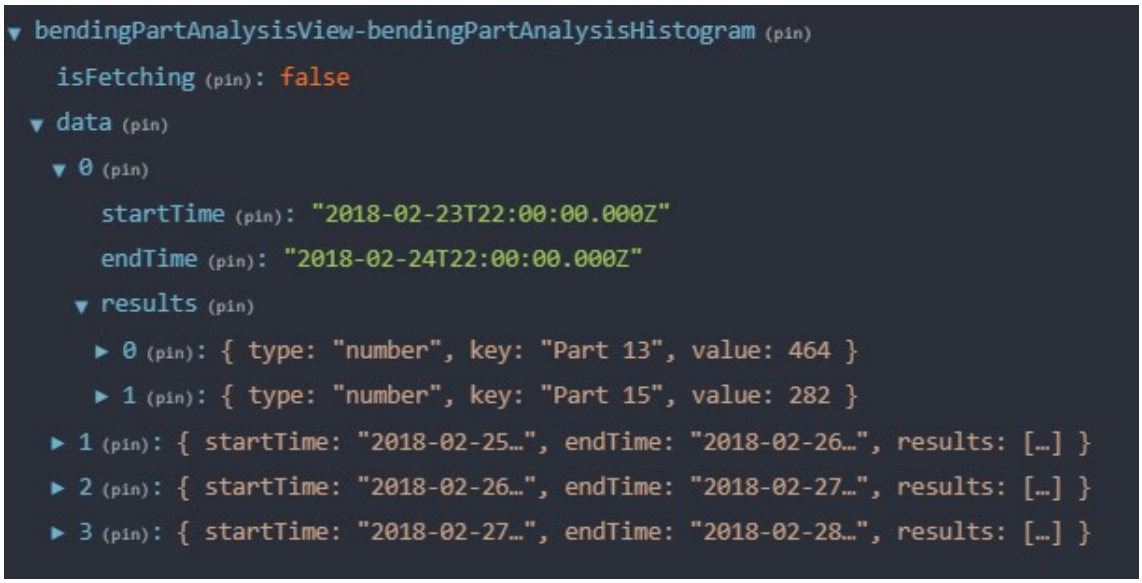

Figure 14. Screenshot of the Redux DevTools utility displaying the Redux state of the “Part count” with the data formatted for the HistogramVisualization component.

The next chapter gives an overview of Reporting Application functionality implemented so far and demonstrates the usage of the dashboard widgets in the context of the various views of the Reporting Application client.

6 Application functionality

This chapter gives an overview of the Reporting Application functionality from the user perspective. The process of setting up the data sources is described, as well as the functionality of predefined and user-defined dashboard views. The chapter also mentions some of the features of the application which are still in development and some issues requiring improvement.

Reporting Application provides a number of dashboard views containing visualizations backed by the consolidated data from the application’s data store. Reporting Application also allows the user to set up data sources and perform historical data import operations by using the application’s GUI. This functionality is provided by one of the settings views.

There are other settings views of Reporting Application that allow the user to set a number of parameters that affect the data search API filters and to configure the localization settings of the application.

6.1 Setting views

The data sources can be configured in the Data source settings view of Reporting Application. To set up a data source the user needs to choose first the type of the data source – either HistoryDB or HistoryFiles. The further configuration of the data source slightly differs depending on its type. In case if the HistoryDB source is selected, the IP address of the MSSQL server to export the historical data from should be provided. In case if the HistoryFiles source is selected, a more detailed configuration for each machine that produced the historical data is required. An example of a data source configuration can be seen in Figure 16.

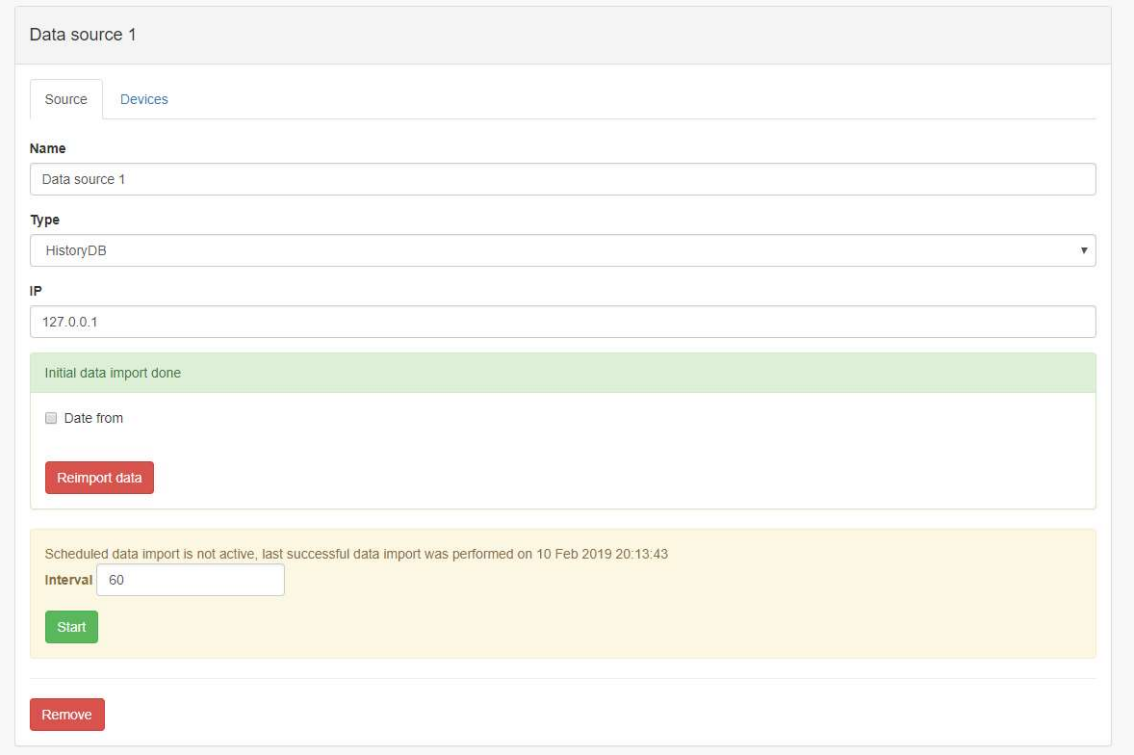

Figure 15. Screenshot of a data source configuration panel from the data sources settings view.

The user needs to know the types of machines that produced the historical data to be able to set up the data source configuration correctly. In case if a HistoryDB source selected the user also needs to know some of the HistoryDB details to configure the export

of the data. A service that will facilitate this configuration procedure is currently in development. In case a HistoryFiles is being configured the user need to set up the file paths for each type of the history files by using the GUI.

Upon settings up a data source the user can start an initial data import, after the initial data is complete a scheduled data import can be set up to keep the datastore in sync with the history data source. It is worth mentioning that the initial data operation could be improved. The initial data import operation is a time and resources consuming operation. At the moment there is no way to cancel the operation from the GUI and in case if the operation was interrupted it should be restarted from the beginning. Besides the need for the operation management implementation improvement, the algorithm behind the initial data operation could be also improved to reduce the required time and the consumption of the resources.

Several settings views are related to the configuration of additional filters that are applied during data search operations. For example, work shifts can be set up to filter out the data from the time periods when the production was idle. It is also possible to set up a global filter that excludes the data that belongs to specific days of the year, this feature allows the user to exclude the irrelevant data from the search.

Reporting Application also provides a Language settings view. In this view, the user can select the language of the GUI and configure the date time and unit of measurement locale settings. The implementation of the client localization logic is almost complete, however, one of the shortcomings of the current implementation is the fact that the translations for many GUI texts are missing at the moment.

6.2 Predefined dashboards

Reporting Application provides a number of predefined dashboard views which display data related to blanking and bending part production, production order details, and machine runtimes. Each dashboard view has a number of widgets displaying various details related to one of the data types. An example of predefined dashboard views could be the bending part analysis view shown in Figure 17. Each predefined dashboard view contains a number of data search related controls. In the center of the view sub-header, a control for selecting a time period is provided. On the right side of the sub-header, there is a control for selecting the length of the time period with possible options of one day, week or month and a control for selecting work shifts which can be configured in the

settings view. The controls listed above affect the search parameters for all widgets of the view.

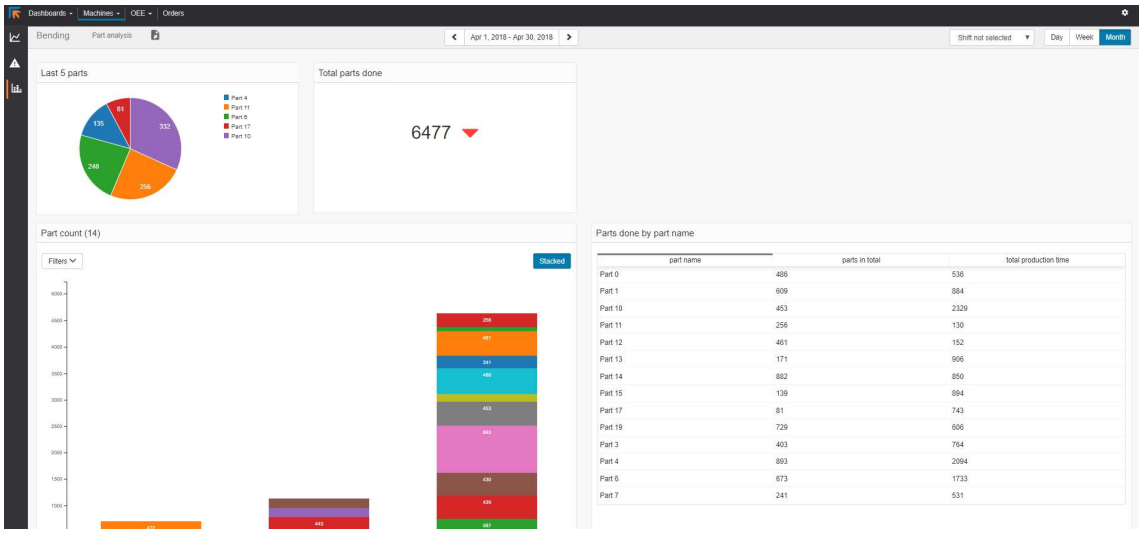

Figure 16. Screenshot of the bending part analysis predefined dashboard view.

The dashboard widgets have a number of controls as well. The widget controls allow to set different visualization options for the widget and in some cases apply addition widget specific data search filters. Some of the widgets also provide navigation possibilities. For example, it is possible to navigate to the bending part details view by clicking on a bar item of the ”Part count” widget from the bending part analysis view. The bending part details view provides details information about the production of a bending part with a certain name. The navigation pattern described above also applies to some other views of Reporting Application, such as nest part analysis, batch analysis, and order analysis view.

The predefined dashboard views’ implementation could be also improved. There are minor GUI bugs to be fixed and the color scheme of the chart visualization could be made more consistent. Moreover, there are few widgets that still lack the data flow implementation, as an additional data source connector should be implemented first. Although the localization logic is implemented for all of the views, the translations for some of the GUI text are still missing.

6.3 User-defined dashboards

Besides the predefined dashboard views, Reporting Application also provides a possibility for the user to create their custom dashboard views. It is possible to create as many custom dashboard views as necessary. Each custom dashboard view supports different layout configurations and allows to add new widgets and set their position. At the moment Reporting Application lacks a user management system and the custom dashboard views are shared between all users of the application. However, there is a plan to add the user management functionality in the future to make it possible for the users to create their personal dashboards and to select the dashboards to be shared with the other users. An example of a custom dashboard view can be seen in Figure 18.

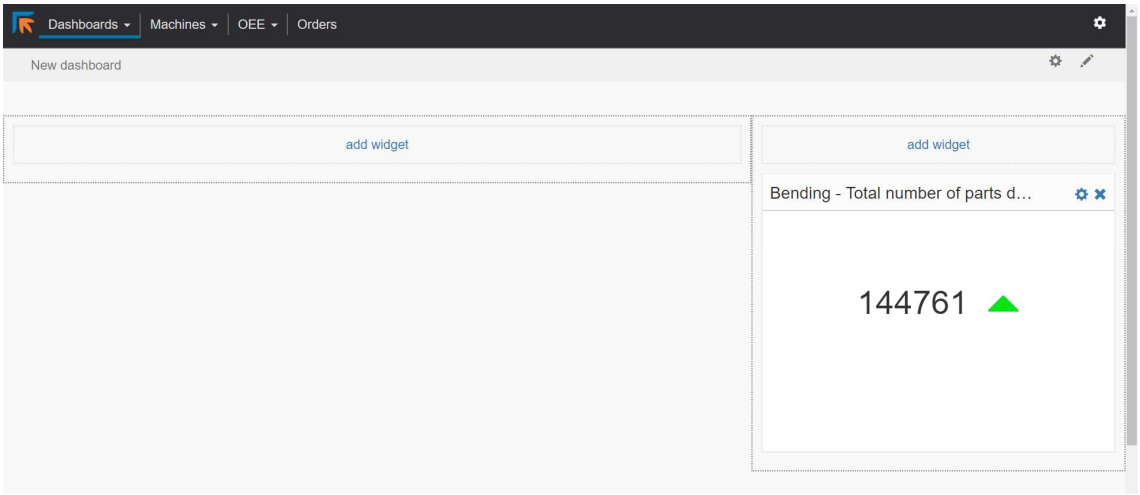

Figure 17. Screenshot of a user-defined dashboard view containing one widget.

The user can add new widgets by a means of the ”Add widget” dialog. The dialog provides options to select the machine and the data type by using dropdown list controls. It also provides a possibility to select a time range for each widget individually. For some of the data types, it is possible to select additional visualization options. For example, the ”Parts done” time histogram bar chart from the bending part analysis view could be visualized as a line chart or a table view.

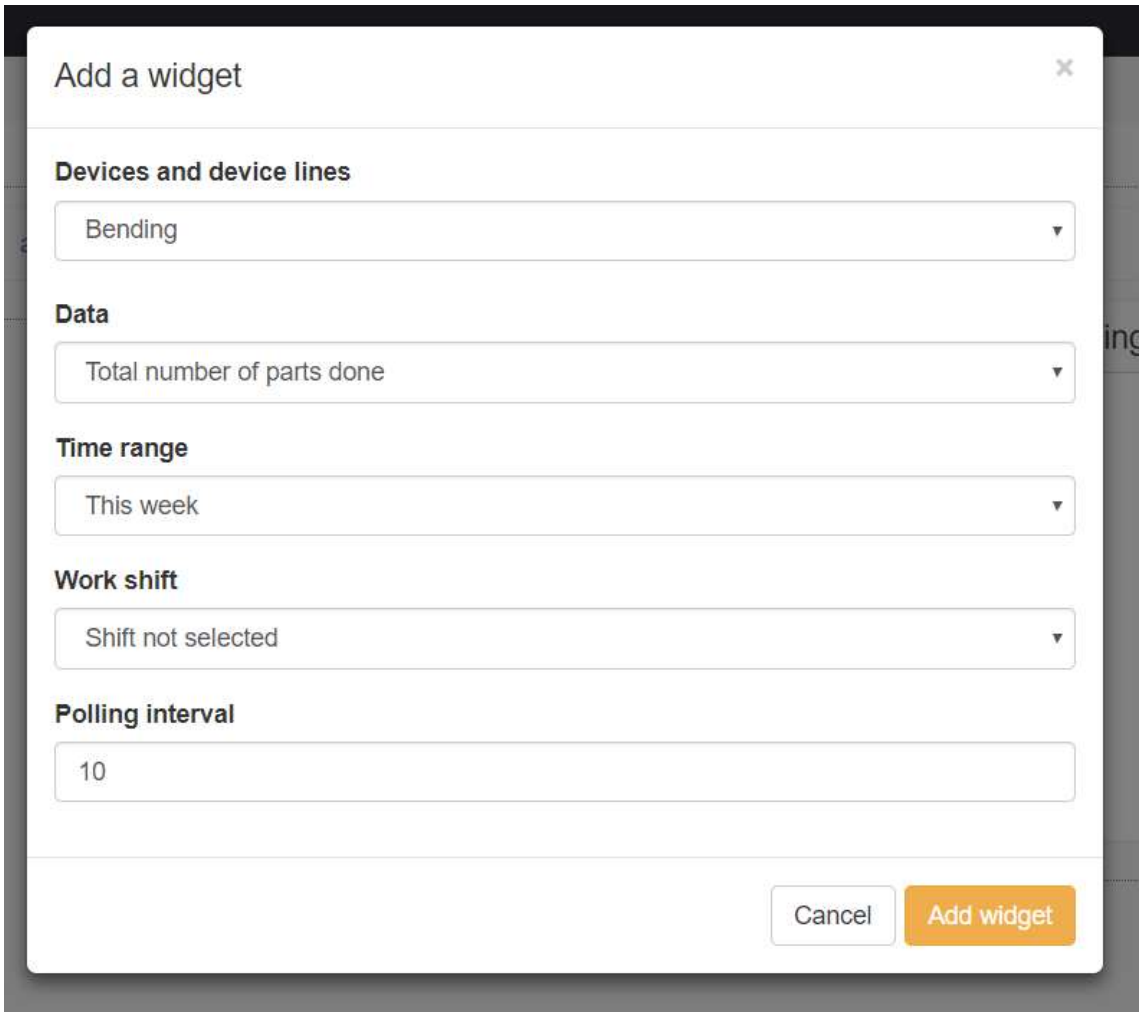

Figure 18. Screenshot of the ”Add widget” dialog.

Overall the user-defined dashboard views complement the functionality provided by the predefined dashboard views. They offer a more interactive and flexible way for analyzing the data and can be used by the user for a variety of purposes, such as performing more in-depth data analysis, creating a custom dashboard containing necessary key performance indicators to display at the factory, or creating a custom financial report. Although Reporting Application still has a number of issues to address, and the implementation of some of the planned features is still missing, all of the core mechanics are already implemented and the application is able to provide most of the required functionality.

7 Conclusion

The main goal of the thesis report was to investigate software development methods and techniques used in the implementation of a web-based reporting application. The main

focus of the thesis report was on the implementation details of the functionality related to reporting and BI elements of the application, such as the data source and datastore management, and most importantly the implementation details of the data search API and data visualization. The detailed description of more trivial technicalities, such as the use of the functionality provided by the .NET Web API framework and Entity Framework, as well as the use of React and Redux libraries, were intentionally omitted so that the more relevant subjects could be explored in more detail. However, the implementation details of the reporting and BI elements were presented in the implementation context of the whole application, and it was shown how the development of these elements benefited from using the functionality of the frameworks listed above.

The thesis report also clearly explained the data flow in a reporting application by providing detailed examples of how the application aggregation functionality works in Section 5.2. Moreover, the detailed examples of the data search API capabilities were provided in Section 5.3. And Section 5.4.2 explained how the client builds the visualizations on top of the data retrieved by using the data search API.

A pilot version of Reporting Application software was installed on a customer's site. The feedback from the customer helped to improve various GUI controls, especially those related to time range and work shift parameter settings. It also helped to enhance the application logic, yielding more relevant data aggregation values. In general, Reporting Application solves a number of issues present in the previous reporting software developed by the company. The Reporting Application solution provides a solid extendable codebase by separating the datastore management, data aggregation, and presentation concerns. In addition, the web-based nature of the application facilitates the distribution of the software. The flexibility of the data search API and the visualization capabilities of Reporting Application make it possible to provide the company's customers with more useful and meaningful information concerning their production.

References

1 Lei H, Yifei H, Yi G. The research of business intelligence system based on data mining. 2015 International Conference on Logistics, Informatics and Service Sciences (LISS) [Internet]. Barcelona, 2015 July [cited date 2019 January 20]; pp. 1-5. Available from: https://ieeexplore.ieee.org/document/7369786/

2 Qihai Z, Tao H and Tao W. Analysis of Business Intelligence and Its Derivative - Financial Intelligence. 2008 International Symposium on Electronic Commerce and Security [Internet]. Guangzhou City, 2008 August [cited date 2019 January 20]; pp. 997-1000. Available from: https://ieeexplore.ieee.org/document/4606219/

3 Gang T, Kai C, Bei S. The research & application of Business Intelligence system in retail industry [Internet]. 2008 IEEE International Conference on Automation and Logistics. Qingdao, 2008 September [cited date 2019 January 20]; pp. 87-91. Available from: https://ieeexplore.ieee.org/document/4636125

4 Herwig V. Business intelligence as a service for Cloud-based applications. 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS) [Internet]. Berlin, 2013 September [cited date 2019 January 20]; pp. 188-192. Available from: https://ieeexplore.ieee.org/document/6662668

5 Microsoft. Data sources in Power BI Desktop [Internet]. Microsoft; updated 2018 October 15 [cited date 2019 January 20]. Available from: https://docs.microsoft.com/en-us/power-bi/desktop-data-sources

6 Qlik Help. About Qlik Connectors [Internet]. QlikTech International AB; [cited date 2019 January 20]. Available from: https://help.qlik.com/en-US/connectors/Content/Connectors_Home/Home.htm

7 Open Source Search & Analytics · Elasticsearch | Elastic. Input plugins [Internet]. Elasticsearch B.V.; [cited date 2019 January 20]. Available from: https://www.elastic.co/guide/en/logstash/current/input-plugins.html

8 Microsoft. SQL Server Documentation [Internet]. Microsoft; updated 2018 August 12 [cited date 2019 January 20]. Available from: https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017

9 Microsoft. Entity Framework 6 [Internet]. Microsoft; updated 2016 October 23 [cited date 2019 January 20]. Available from: https://docs.microsoft.com/en-us/ef/ef6/

10 React – A JavaScript library for building user interfaces [Internet]. Facebook Inc. [cited date 2019 January 20]. Available from: https://reactjs.org/

11 Redux · A Predictable State Container for JS Apps [Internet]. Dan Abramov and the Redux documentation authors; [cited date 2019 January 20]. Available from https://redux.js.org/

12 C3.js | D3-based reusable chart library [Internet]. Masayuki Tanaka; [cited date 2019 January 20]. Available from: https://c3js.org/

13 vis.js - A dynamic, browser based visualization library [Internet]. Almende B.V.; [cited date 2019 January 20]. Available from http://visjs.org/