bachelor's thesis

Design and prototype implementation of a module for graphical representation of data flow parser

Yehor Petukhov

May 2020

Ing. Michal Valenta, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Graphics and Interaction

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Petukhov Yehor**	Personal ID number:	**466364**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Branch of study:	**Computer Games and Graphics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Design and prototype implementation of module for graphical representation of data flow parsers

Bachelor’s thesis title in Czech:

Návrh a prototypová implementace modulu pro grafickou reprezentaci toku dat v analyzátorech

Guidelines:

Manta develops a set of tools for analyzing and visualizing of data flows (data lineage) in large software systems. The aim of this work is a prototype module that will be used by developers specializing in the design and implementation of syntactic analyzers of various languages (SQL, Pig, Hive, Kafka, OBIEE, ...) for the gradual development and testing of syntactic analyzers.
Follow these steps:
1. Review the Manta Toolkit.
2. Analyze requirements for graphical tool with domain experts (parser developers).
3. Design the graphical interface of the tool and implement its prototype.
4. To verify the design, implement your own data flow parser for the Microstrategy reporting system and test the usability of your solution.
Implementation platform: javascript, typescript.

Bibliography / sources:

[1] William Lidwell: Universal Principles of Design, Revised and Updated. Rockport Publishers Inc., ISBN: 9781592535873, 2010.
[2] Joshua Bloch: Effective Java. Third Edition. Pearson Education. ISBN:9780134685991, 2017.
[3] Uday Khedker, Amitabha Sanyal, Bageshri Karkare: Data Flow Analysis,Theory and Practice. CRC Press, ISBN 978-0-8493-2680-0, 2009.

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Valenta, Ph.D., Department of Software Engineering, FIT

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **11.02.2020** Deadline for bachelor thesis submission: __________

Assignment valid until: **30.09.2021**

	[illegible]	[illegible]
Ing. Michal Valenta, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

Acknowledgement

I want to thank Michal Valenta for the help and support provided while writing this work. I also want to thank the people who spent their time helping me with their expert opinion - Petr Kosvanec and Yauheniy Buldyk. Furthermore, the main thanks to my family, which always supported me.

Declaration

Abstract

Jak sledovat veškerý tok dat v obrovském množství dat? Pro tento účel existuje speciální software, například Manta Flow. Účelem této práce je implementace nového modulu pro software Manta Flow, který je zaměřen na analýzu reportovací platformy MicroStrategy a tvorbu grafů představujících datové toky uvnitř ní. Druhou částí této práce je vytvoření funkčního prototypu nástroje s uživatelským rozhraním na základě zkušeností získaných při vývoji analyzátoru. Nástroj musí zjednodušit budoucí proces vývoje nových modulů. V poslední části práce musí být ověřena použitelnost nástrojů.

Klíčová slova

Tok dat, uživatelské rozhraní, vývojářské nástroje, Manta

Abstract

How to track all data flow in a massive amount of data? There is a special software for this, for example, Manta Flow. The purpose of this thesis is to implement a new module for Manta Flow software, which is aimed to analyze MicroStrategy reporting platform and produce graphs representing data flows inside of it. The second part of this work is to create a functional prototype of a tool with a user interface, based on the experience gained in the development of the analyzer. It must simplify the future process of new module development. As a last part of the work, tool usability must be verified with domain experts.

Keywords

Data flow, user interface, developer tools, Manta

Contents

Abbreviations

REST	REpresentational state transfer
API	Application programming interface
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
HTML	Hypertext Markup Language
JS	JavaScript
UI	User interface
UX	User Experience
SDK	Software development kit
ETL	Extract, transform, load
MVC	Model–view–controller
CSV	Comma-separated values
XML	Extensible Markup Language
JSON	JavaScript Object Notation
OLAP	Online analytical processing
DOT	Graph description language
CLI	Command-line interface
ID	Identifier
OBIEE	Oracle Business Intelligence Suite Enterprise Edition

1 Introduction

Nowadays, data is one of the essential resources, especially for companies of all sizes. We process data, create it, store, and spend a massive amount of money on protecting it from competitors. In this project, we will focus on data analysis.[1]

Based on data analysis, companies can get information about business direction, employers statistics, and much more internal and external stuff. With introducing big-data, things are getting even more interesting. But there is one crucial problem - it is beyond human capabilities to process such an amount of data, especially if we need speed and accuracy. One solution is to use particular tools that can transform raw data into human-readable form, e.g., table, graph, diagram. This sort of software is called "reporting tools" and used to analyze internal and external data to make business decisions. It is mostly used by business analytics, to save time and be more productive by visualizing data.[2]

There are various products to choose from (Oracle Business Intelligence Suite Enterprise Edition (OBIEE), Cognos, Tableau), but we will focus on the MicroStrategy platform. This powerful instrument, among other functionalities, provides data analyzing. But there is a common omission in this software segment. There is no possibility to ensure data correctness and track all data flows, from sources to the final report.

This is what Manta's software does. Application process data flow in many reporting tools by extracting metadata, analyzing, and preparing visual output in the form of a graph in their application. There is a separate module for every tool that is supported by Manta Flow since the internal structure of all reporting tools is almost unique. It means that a programmer every time should precisely map extracted metadata structure to internal graph structure while writing a module for new technology, to be able to work with data in Manta Flow. Because of complicated architecture, this process is somewhat entangled. This is the primary problem that will be solved in this project.

1.1 Goal definition

The main goal of this project is to design and implement a functional prototype of a module for data extraction and analysis to expand Manta Flow software. Prototype functionality must provide the possibility to process data from MicroStrategy tool and generate a graph that will show all data flows between data sources and MicroStrategy's final result. As a key detail, the module must be tested and documented like a regular software project.

After getting experience with writing this type of software, and analyzing existing help tools that are used by programmers, discuss the result of analysis with domain experts, design a prototype of the tool that will provide useful features to help the development process. The tool with User interface (UI) must fulfill requirements, which will be defined by experts (users). It can be a replacement for existing tools or a completely new instrument.

1.2 Plan

- Get used to the MicroStrategy platform to understand all key features and capabilities of the software. Document expected behavior and possible artifacts. Dissect software endpoints, to find out a way to get all needed information from the platform.
- Implement the main Manta Flow module. It consists of `Extractor`, `Resolver`, `Model`, `Generator`, `Analyzers`. Will be described later in detail.
- Analyze the existing environment and design the prototype of a new tool.
- Verify that a new tool solves a problem, described by domain experts.
- Make conclusions about future work.

2 Used technologies

2.1 Data flow parser

2.1.1 Java

Java is an object-oriented, class-based language that is platform-independent and can be run on any platform that supports Java Virtual Machine. Java is commonly used in various projects for both Desktop and Web, target platforms. Manta Flow is written in Java, and because of it, a new module for MicroStrategy also will be done in this language. The reader doesn't need to have a good knowledge of Java, to understand everything described in this work.[3]

2.1.2 Maven

Maven is a build automation tool used for Java projects. Will be used to automate build scenarios, manage libraries and artifacts.[4]

2.1.3 JUnit

JUnit is a framework intended for unit testing in Java projects. Together with Maven will ensure quality automatic testing.

2.1.4 JavaDoc

JavaDoc is a documentation generator for generating Application programming interface (API) documentation in Hypertext Markup Language (HTML) format from Java source code.[5]

2.1.5 Spring

Spring is an application framework and inversion of control containers for the Java platform. It used to benefit from the capabilities of Spring Bean.

2.1.6 Git

Git a primary version-control system for tracking changes in source code during software development.

2.1.7 Intellij IDEA

Intellij IDEA is an integrated development environment for developing Java software. Free Community edition will be used.

2.1.8 Postman

Postman is an API client tool that helps to test the APIs. It makes it possible to test the same request against different environments with environment-specific variables. It is used to find and test MicroStrategy REpresentational state transfer (REST) points.

2.2 Graphical tool

2.2.1 TypeScript

TypeScript is an open-source programming language. It is a strict syntactical superset of JavaScript and adds static typing to the language. Language is designed for the development of all-size projects and transcompiles to JavaScript, which means that existing JavaScript programs are also valid TypeScript programs. Such a choice is explained by the fact that static typing makes JavaScript closer to traditional to us languages, such as C, C++, C#, and Java.[6]

2.2.2 React

React is a JavaScript library for building user interfaces. Chosen because this is one of the most popular frameworks for this purpose, is greatly documented and maintained by a big company (Facebook), which means that React is relevant and will be supported for some time. One of the most significant advantages is that it has a low entry-level.

2.2.3 React-diagrams

React-diagrams is an open-source TypeScript library for building interactive diagrams. React-diagrams offers many advanced features for user interactivity such as drag-and-drop, copy-and-paste, in-place text editing, context menus, data binding and models, event handlers, commands, and customizable animations.

2.2.4 Blueprint JS

Blueprint JS is a React-based UI toolkit for the web. Used for faster and easier web development, allowing building graceful user interfaces. Lightweight and straightforward to use.

2.2.5 Webpack

Webpack is an open-source JavaScript and TypeScript module bundler.

2.2.6 Git

Git a primary version-control system for tracking changes in source code during software development.

2.2.7 Visual Code

Visual Code is a development environment, among other things, used for JavaScript and TypeScript development.

3 MicroStrategy Analysis

MicroStrategy platform consists of two main components that will be described on the next pages. We will also analyze the REST API and platform artifacts.

3.1 MicroStrategy Full Platform [7]

`MicroStrategy Full Platform` - architecture consists of four main elements: `Metadata`, `Warehouse`, `Intelligence Server`, and `Web Server`. It is the basis without which the full-fledged work of the program is impossible.

- `Warehouse` - database which contains the information that users work with, some sort of internal disk. E.g. Comma-separated values (CSV), Extensible Markup Language (XML), JavaScript Object Notation (JSON), data extracted from databases etc. This information is usually placed or loaded in the data warehouse using some sort of (Extract, transform, load (ETL)) process.

- `Metadata` - is a repository that stores MicroStrategy object definition and `Warehouse` data usage information. The metadata allows the sharing of objects inside of the MicroStrategy environment. All data are stored in a proprietary format.

- `MicroStrategy Intelligence Server` - analytical server optimized for Online analytical processing (OLAP) analysis, which performs such important functions as executing reports, dossiers, and documents against the data warehouse, objects, and data sharing, objects management. Makes all calculations, works with data. All data is saved as a path to the data source or stored in special cubes. A data cube is a multi-dimensional array of values used to represent data. The server provides Software development kit (SDK) and a rich collection of REST points (as a separate `MicroStrategy REST Server`), that is ideal for our purpose.

- `MicroStrategy REST Server` - consumes the incoming Hypertext Transfer Protocol (HTTP) request, converts it into an XML command and passes the XML command to the `Intelligence Server` that generates the requested results, and passes them back to the `MicroStrategy REST Server`.

- `Web Server` - service that provides users with an interactive environment to access and analyze data via web-browser, view file system on the server.

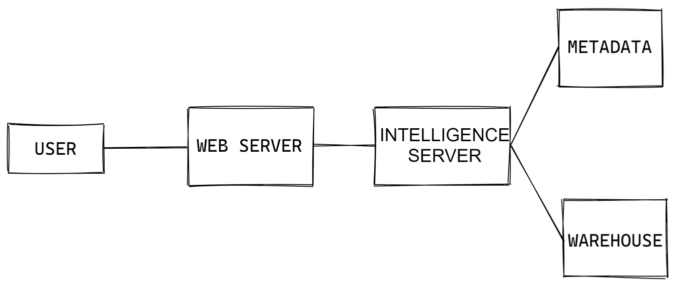


Figure 3.1 A simplified version of MicroStrategy architecture

3.2 MicroStrategy Desktop

`MicroStrategy Desktop` is a free desktop application that is created to get experience with the product without additional configuration. It is simple to install and use. It doesn't have any dependencies. It also includes all common user features. Allows creating three types of data analysis: Report, Document, Dossier (3.4). `MicroStrategy Desktop` supports export to all popular formats (e.g., pdf, png, CSV, etc.). It doesn't provide any endpoints. A perfect solution for regular users, but doesn't fit developer requirements. However, it can work both independently and directly with the `Intelligence Server`, as a replacement for `Web Server`.

3.3 MicroStrategy REST API [8]

MicroStrategy API is a RESTful application that provides a way to extract all data related to dataflow that will be analyzed, from `Metadata` and `Warehouse`. Data is returned as a response to an HTTP request, in the form of JSON. This will allow us to extract data without interfering with the internal structure of the program. Such a process can be done via multiple API families - groups of the related access points (Authentication, Browsing, etc.). In the scope of the analyzer prototype, we are interested in the next families:

- Authentication API - authenticate a user and returns an authorization token (X-MSTR-AuthToken), which will be included in future requests.
- Projects API - return a list of all project Identifier (ID)s, which we have access to.
- Browsing API - after providing authorization token and project ID, return a list of file IDs stored in this project, and matching search filter.
- Dossiers and Documents API - return high-level specification of dossier / document without any data. Also, allow getting cube IDs related to any dossier / document.
- Cubes API - return specified cube raw data without direct formatting.

3.4 MicroStrategy artifacts

In `MicroStrategy Desktop` and `MicroStrategy Web` can be created three primary sorts of objects:

- Report – is a MicroStrategy object that represents a request for a specific set of formatted data from your data source. In its most basic form it consists of two-part:
  - A report template, which is the underlying structure of the report.
  - The report-related objects are placed on the template, such as attributes, metrics, filters, and prompts.

  It is the most basic object that can be created, has a relatively simple structure, and allows us to show only one unit of data at a time. It doesn't have any interactive features, can only display or export simple data visualization.[9]

- Document – contains objects representing data coming from one or more MicroStrategy reports, as well as images and shapes. Documents can appear in many ways and are generally formatted to suit business needs. A sample document is displayed on the right.Because a document consists of more reports, it provides a possibility to create various, complex data visualizations.[10]

- Dossier - is the most powerful interactive display that you can create to showcase and explore data. It is possible to add simple visual representations of the data (called visualizations) to the dossier to make the data easier to interpret, perform manipulations on the data to customize which information to display, organize data into multiple sheets and pages to provide a logical flow to the dossier. Allows quickly and easily create a polished dossier without requiring a lot of design time using visualizations and predefined formatting.[11] Example dossier on Figure 3.3

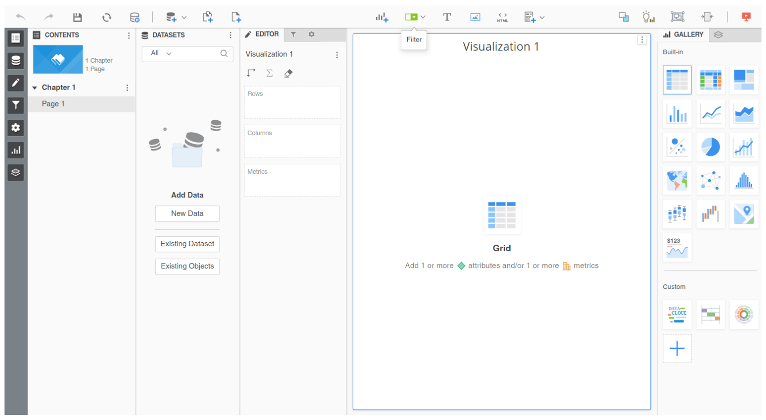


Figure 3.2 Dossier editor

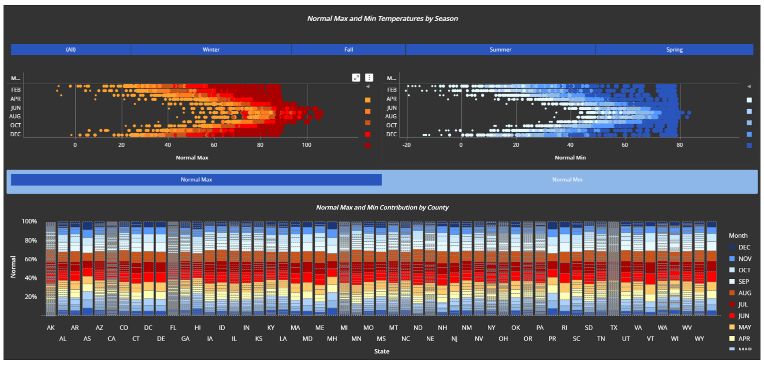

Normal Max and Min Temperatures by Season
(All)
Winter
Fall
Summer
Spring
Normal Max
Normal Min
Normal Max and Min Contribution by County
Normal
State
Month

Figure 3.3 Dossier example[1]

[1]Source: https://www.microstrategy.com/us/resources/library/dossiers

4 Data flow parser implementation

After learning existing `Manta Connectors` and documentation, it has been found that to visualize the data flow of the MicroStrategy platform, the data flow parser must go through the following steps:

1. Analyze the entire MicroStrategy file system and find all objects that are important for data flow analysis.
2. Download these objects and save them on disk as files.
3. Load the files from the disk and analyze them.
4. Create a `Model`(4.2) of Java objects that will represent and describe the analyzed file system.
5. Create a graph from the `Model`(4.2), which will be used for visualizing data flows.

The whole module is divided into three main parts, each performing its tasks. The `Extractor`(5.1) is responsible for the 1st and 2nd steps, the `Resolver`(4.2) takes care of the 4th and 5th steps, and the last step is a task of the `Generator`(4.3).

The connector is made in such a way each part of the parser is an independent functional unit, which is able to operate without others. So it allows us to implement each part of the module separately and test it before completing the overall prototype.

Such architecture also allows clients to avoid using `Extractor`(5.1) in the client environment if it contains sensitive data, or it is a highly insulated system. In this case, artifacts of `Extractor`(5.1) run must be prepared manually, to ensure the operation of further components.

Orchestration of all these parts is done via scripts and Spring, from the Manta CLI program (4.3).

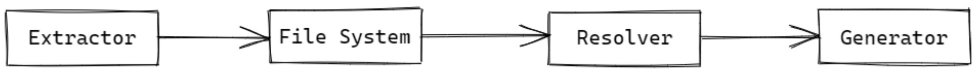


Figure 4.1 Data flow parser structure

4.1 Extractor

The main goal of the `Extractor` is to download all necessary files from the MicroStrategy server and save them on the disk.

We can get all objects in the form of JSON using the available REST API and Apache HTTP library. Used REST points were described above (3.3), and because of the risk of unexpected updates from MicroStrategy, it is important to keep this part of the project highly extendable and flexible. With this idea in mind, was created a structure where each key element (dossier, document, cube, etc.) has its own `ElementWorker` and `ElementRequestManager`.

- `ElementWorker` - contains the main logic of element processing and order of REST requests that must be called.
- `ElementRequestManager` - provides simple methods that execute predefined API requests and return JSON answers from `MicroStrategy Intelligence Server`. Also extends `AbstractManager`, that grants access to unified authorization method. All authorization credentials are passed in the form of a configuration file (Uniform Resource Locator (URL), username, password).

After receiving all data we can store them on a hard-disk as a file, but this Manta Connector is designed to serve as a corporate solution, it will process vast amounts of data, so it is necessary to use some space optimization to avoid writing gigabytes of data to disk.

In the future dataflow, we will not need raw data, it is enough to have their "definition" or "specification". Accordingly, we can save a lot of space by removing redundant information from JSON. To achieve this was used Jackson - JSON processor for Java that allows working with JSON as with simple Java objects. After deleting unnecessary data, we proceed to save the rest.

Using JSON, we deserialize Java classes into regular text and store reports, documents, dossiers, and data sources as files in a file system.

The result of the `Extractor` work is a directory, which contains files with metadata of MicroStrategy objects. This directory and files are used as input for the `Resolver`(4.2).

4.2 Resolver

`Resolver` module contains three main parts:

- The first one is the `Input Reader`, that reads a raw data from the disk.
- The second part is the `Model`, which represents a collection of Java interfaces used for a detailed description of objects needed for data flow visualization.
- And the last part is `Resolver` itself, which is just an implementation of `Model`'s interfaces.

`Input Reader` - actually consists of two readers - `SequenceFileReader` and `DiscreteFileReader`:

- `SequenceFileReader` consumes text data from the file system, but because of the pretty complex relationship between extracted data - we look only for dossier(3.4) and document(3.4) definition for now. Each found definition we pass to the `Resolver`.
- `DiscreteFileReader` - takes the path to a file as an input parameter, reads data, and passes it to the `Resolver`. A key feature of this reader is a cache that allows reading less from disk, considering that one file can be requested many times.

The main difference from the first reader is that `DiscreteFileReader` is an iterative reader, whereas `DiscreteFileReader` looks for an exact file. This is used later in the `Generator`.

`Resolver` - main logic of this part is to take the input file and, according to its type, maps it to appropriate `Model`, using a complex system of constructors in `Model` implementation.

`Model` – is a bunch of interfaces and implementations reflecting an internal representation of extracted data. Only the `Model` is used in the Generator as an API to access data. So we separate interfaces and their implementations to increase encapsulation and sustainability of the code.

So as a result of this module run, we have a model of java objects representing real objects from the MicroStrategy server, which can be used for generating an output graph of data flow.

4.3 Generator

The last part of the parser is the `Generator`, whose sole task is mapping java objects from `Resolver`(4.2) to nodes of the data flow graph. To configure and run this module Spring framework and configuration files are used. `Resolver`(4.2) is passed as a parameter to the `Generator` to use both provided readers (described above 4.2). Firstly, the algorithm maps objects returned by `SequenceFileReader` to nodes of Manta Flow visualization. After this is done, it pulls out all the data about the relationships with the data sources (cubes) and asks for these objects from `DiscreteFileReader`. The received object is also mapped to nodes and merged with an existing graph.

Mapping to nodes is on special classes - `Analyzers` - that analyze input objects, create corresponding nodes in the graph, and add data flows between them. The `Generator` also uses different existing services for building different parts of the graph (e.g., data flow of a database used as a data source in some report) and then connect them all together. `Analyzers` also use the help class `MicroStrategyGraphHelper`. This class is used to build the resulting graph and keep its current version during creation. Probably the most important and used method in this class is `buildNode`, which can build and place a node to the right place in the existing graph, based on an input object of type `MicroStrategyItem`.

At the same time, this method recursively creates all ancestors of the node up to the root. That means that we can send to the `buildNode` method only the leaves of `MicroStrategyItem` without worrying about its hierarchy. This approach allows us to implement a smart process of adding nodes to graph, it takes care of already existing nodes and adds only missing part.

In the end, we have the output graph, which consists of nodes and flows between them, and represents the whole data flow from data sources to report items of each object from the MicroStrategy server. As a next step, we send this graph to Manta CLI, which will visualize it with many applied filters. This graph is the most important thing for our future tool and is the definition of done for data flow parser.

Manta CLI - is a set of scripts and scenarios that serves as an interlayer between the connector and Manta visualization. It accepts configuration files, invokes the Spring context, and decides whether to call only `Generator` or first call the `Extractor`(5.1). CLI is also responsible for passing the resulting graph from `Generator` to sisualization. This part of the workflow is already done in MicroStrategy and is just a target for connector integration.

4.4 Conclusion after implementing Data flow parser

The main aim of this part of work is to design and implement a functional prototype of a data flow parser for analyzing and processing MicroStrategy data streams

- to get into the working process, implement complex software adhering to all the rules of writing high-quality product.
- to be able to investigate and propose a tool with a user-friendly user interface that will simplify the future process of writing data flow parsers in Manta.

As a result, analysis of the MicroStrategy reporting tool was conducted and examined the existing possibilities of processing its objects. Based on the analysis, was designed and implemented a module that can download specific MicroStrategy documents from the server, then process them and convert them into a Java model, from which it is easy to generate a graph to visualize data flows (Figure 4.2). The codebase is covered with automated tests and javadocs. The project was also manually tested and successfully integrated into the Manta Flow product.

Some statistics:

- 4958 lines of Java code
- 606 lines of XML
- 90 classes

What was not done yet, but will be added in the future:

- Support for all types of MicroStrategy objects, such as Document and Report.
- Support of a database as a data source.

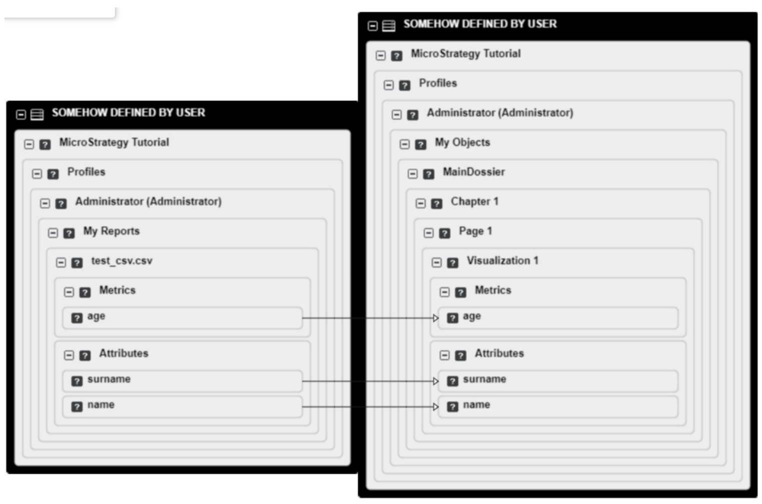


Figure 4.2 Example of MicroStrategy data flow parser in Manta Flow

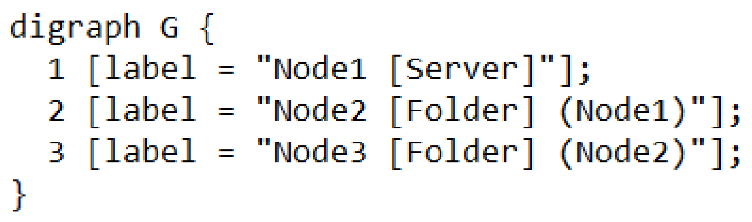
5 Dev-Viewer analysis

After implementing a functional prototype of a Manta Connector (4) and having a conversation with domain experts, it was decided that the most necessary tool with user interface (further just a "Dev-Viewer") is a replacement for the archaic Graphviz program. There are a bunch of needed features that can simplify the development process.

Graphviz is a package of open-source tools for drawing graphs specified in Graph description language (DOT) language scripts. It also provides libraries for software applications to use as the plug-in tool.

There is a special script in the Manta Flow project, that takes the Manta's internal representation of data, extracted from a reporting tool (e.g., MicroStrategy) and converts it to DOT language. After that, a new DOT language script is saved to the file system as a regular file. Finally, Graphviz consumes this file to produce an image of a graph, representing the original Manta Flows internal data structure. This whole complex process is done just to verify manually that mapping of extracted data to internal representation is correct. This can be done dozens of times per day, and this only exacerbates the complexity of the process. Even though the complexity, the result is not suitable for comfortable work, it is just static, not interactive image, quite often with a poor scale. Example of DOT language (5.1) and a Graphviz result image (5.2).

Eventually, it was decided to create a lightweight application that will replace outdated Graphviz. The reason for writing the whole data flow analyzer is to find a problem and come to the right User Experience (UX) design for this tool, through getting real experience.



```
digraph G {
  1 [label = "Node1 [Server]"];
  2 [label = "Node2 [Folder] (Node1)"];
  3 [label = "Node3 [Folder] (Node2)"];
}
```

Figure 5.1 Example of dot format

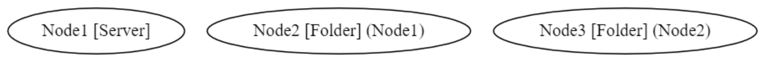


Figure 5.2 Example of how Graphviz displays dot file

5.1 Requirements

To find the right design and implement meaningful functionality, it is necessary to define requirements and definition of done. It was decided that a new graphical tool must visualize the interstate of `Generator`(4.3) run, using an already existing mechanism of export to DOT file. This interstate is actually a graph that represents flows of data inside of some environment, for example, MicroStrategy.

After several rounds of discussions with a manager, and developers that are interested in this sort of software, the following requirements were established:

- Dev-Viewer must be an application that does not need any installation or configuration. This can be solved by using a web solution - an application that runs in a browser.
- Dev-Viewer must provide a human-readable format of data representation, and high interactivity will help solve this problem.
- Display all available data without any filtration due to the fact that it is a development tool, and it is important for debugging, even if Manta Flow hides this information.
- Feature of filtering visualization by node type, but this must be disabled by default.
- Search feature that will find a node by name.
- Export to the image, to maintain old functionality.

The appearance of Dev-Viewer and the rest features were not defined, freedom of choice was granted, because of experience that was gained earlier during the creation of data flow parser.

5.2 Graphical representation

After determining the requirements, work on visualization was started. Many visualization options have been tried, to come up with a correct representation of a graph that is a file system structure with connections between files, and additional attributes.

Examples of tested visualizations 5.3, 5.4, 5.5.

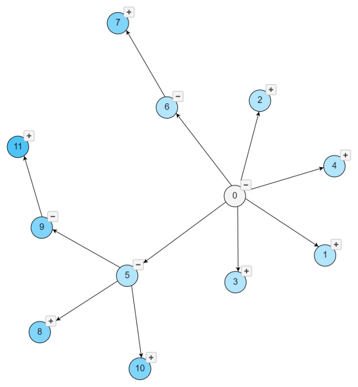


Figure 5.3 Tree[1]

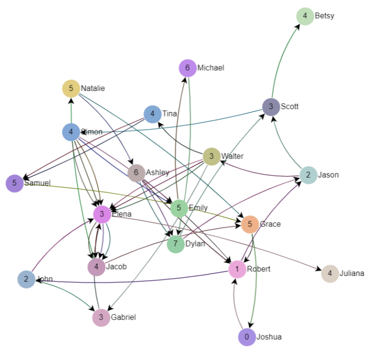


Figure 5.4 Graph[2]

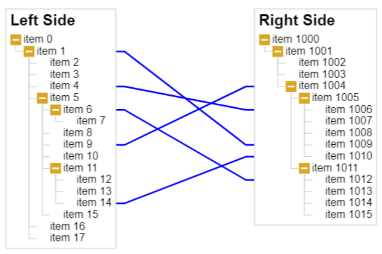


Figure 5.5 Tree map[3]

[3]Source: https://gojs.net/latest/samples/incrementalTree.html
[3]Source: https://gojs.net/latest/samples/distances.html
[3]Source: https://gojs.net/latest/samples/treeMapper.html

There are two ways to implement graph visualization.

- Write new library for displaying data structures in JavaScript or TypeScript. This is the right approach if you have the experience of developing web applications and are familiar with these languages. This will allow fitting all requirements perfectly but will take significantly more time and probably will go beyond the scope of this work.

- Use an existing solution that best suits the required functionality. This way has several disadvantages and limitations. For example, it must be open source to be used in business and ensure that this solution is safe. Also, this library must have at least some maintenance with documentation provided. The existing solution will save a lot of time and allow to start work on Dev-Viewer immediately without further delay, but the price is a higher time of adding new features in the future because the core of the library must be updated to provide expected functionality.

Because of the lack of experience with this type of development, it was decided that the second option is more suitable in current work. After little research was finally found a library that fits best - “react-diagrams v6.1.1”. It is used to create customizable orientated diagrams simply. Unfortunately, it does not support nested nodes, which are great for file structure representation, but this can be solved by using a descending structure with the right offset.

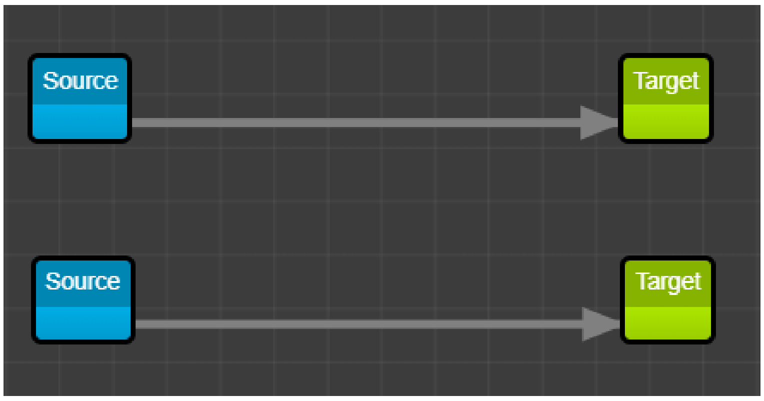


Figure 5.6 Simple example of react-diagrams functionality[4]

The best solution that was found to display the graph, taking into account the selected technologies, is to split the graph based on the relationships between the leaves (Element 5.7). It allows us to keep the visualization clean and readable. This idea is already used in Manta visualization, which is good, this will maintain consistency between the final product and developer tool.

[4]Source: http://projectstorm.cloud/react-diagrams/

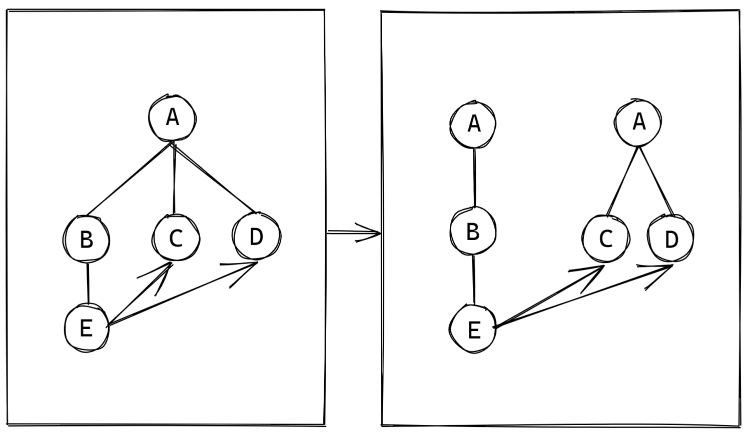


Figure 5.7 Example of graph representation

6 Dev-Viewer implementation

6.1 Architecture

For Dev-Viewer implementation was selected TypeScript because it provides static-typing and selected react-diagrams library is also written in this language. To work with UI was selected React framework for simplicity and maintainability.

There are multiple popular architectures intended for this technology stack, such as Model–view–controller (MVC) and Flux.

- MVC - is a great classic software design pattern that divides user interface from the data model, has the ability to provide multiple views, has better code maintenance. However, it has the following disadvantages: an increased complex setup process, changes in the model or controller force us to rework almost the whole project.
- Flux - is an advanced MVC design pattern, designed to be used in React applications. Compared with the MVC, it has a more complex structure and the connection is unidirectional instead of bidirectional.

Based on this information was selected a simplified version of MVC architecture, where Model and Controller are actually united. That was done to simplify the structure of this relatively small project. The model that represents data is created once and stays immutable until the end of the run. The functionality of the user interface and controller is provided by React itself. So it is MVC, but without explicitly separating View and Controller into different classes, this is delegated to React framework.

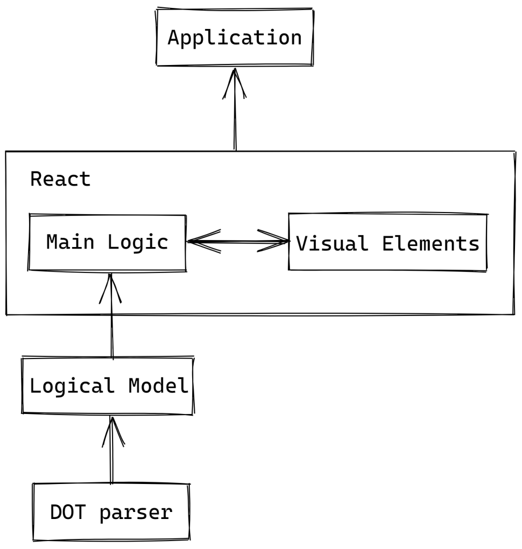


Figure 6.1 Dev-Viewer architecture

As a regular web page, Dev-Viewer starts from `index.tsx`, it is an entry point of the whole application. After that `diagram.tsx` is called, this is the root of all visual elements and contains main event handlers that trigger data procession when the target file with DOT language is uploaded (`handleFileLoad`). Some of the elements have their own event handlers because they do a logically independent piece of work, for example, class `NodeDetail` (Information panel in 6.4). But together with `diagram.tsx` they inherit React functionality for this reason such behavior does not break adopted architecture. That means that there is a program basis that handles main functionality and visualization. This basis is easily extensible with new elements if needed. Such a simple project structure allows us to add new functionality, with zero price initial setup quickly.

To fulfill structure understanding, let's take a look also on the page(Figure 6.2). In general, the user interface consists of the following elements:

- Main working area
- Individual nodes
- Tool panel with buttons

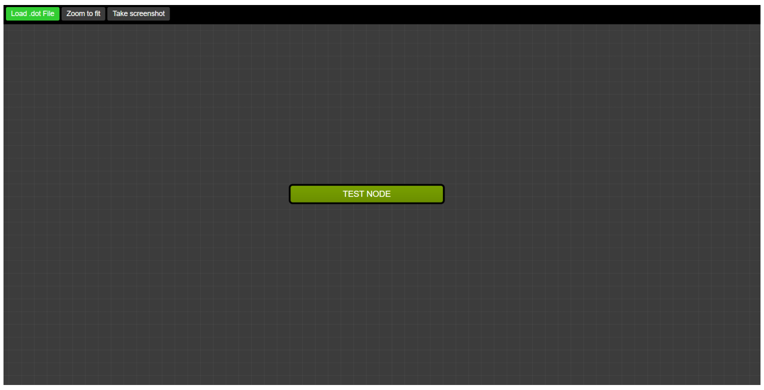


Figure 6.2 Dev-Viewer working area

For now, only the green button "Load .dot File" is important, other buttons will be overviewed later. This button is the start of the usage scenario, it opens a file selection window for uploading, where you have a predefined filter for DOT files. After the user chooses a file, processing starts, this causes Model creation and subsequent visualization. Each file upload is equivalent to application restart. It helps to avoid problems of the inconsistent state.

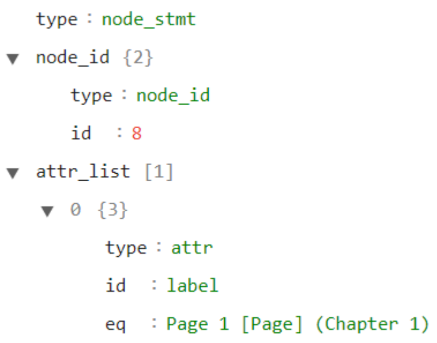
6.2 file processing

After the file upload is finished, one of the most essential processes starts - converting the DOT file into visualization. This is a multiple steps process, that will be described below in details:

1. The file is uploaded and converted into regular JSON, using the dot library. JSON is great for our purposes because it is JavaScript type - natively supported by JavaScript and TypeScript languages. A disadvantage of this conversion is that this conversion stores a label as one attribute. For example, one line of DOT file:

   ```
   8 [label = "Page 1 [Page] (Chapter 1)]"
   ```

   will be stored as (Figure 6.3)

   ```
   type : node_stmt
   ▼ node_id {2}
       type : node_id
       id   : 8
   ▼ attr_list [1]
       ▼ 0 {3}
           type : attr
           id   : label
           eq   : Page 1 [Page] (Chapter 1)
   ```

   Figure 6.3 Example of dot file storing

2. The resulting JSON is passed to `CustomTreeBuilder` that maps the received JSON to the custom tree. This process also includes label splitting into different attributes. The algorithm starts from the root and adds nodes one by one. A simple cache is implemented to avoid onstantly searching for newly created nodes when looking for a node's ancestor. Each node is an object of `Node` class, in addition to the convenient shape of the tree, it also allows us to store real data flow connections between nodes, this will simplify future work.

3. The next step is to call `SubGraphBuilder` - this class transforms one tree to the separated subgraphs representation, which was described above. It works as a black box - just pass the root `Node`, and the build method will return a final result that consists of a bunch of subtrees with connections between them. All links go from left to tight, without circle dependency, it is guaranteed by MicroStrategy architecture. We will take a look at this important class in 6.3.

4. And the last thing to be done is converting all nodes from the internal representation to library nodes - `GraphNodeModel`, that will be visualized. This is done in `ModelBuilder`, thanks to a convenient on-site representation, this is a relatively simple process. Data flow connection is added when a node with outgoing relation is found. Mapping is done from the right to the left side of the graph, to create all the connections between nodes on the fly. Each created visual node we add to the library model. A library model is an object containing all elements that need to be displayed.

5. The model is passed to the render engine.

6.3 SubGraphBuilder

The algorithm implemented in `SubGraphBuilder` solves one of the critical problems, converting an unreadable tree with many intricate connections into the main target of this work - user-friendly interactive representation. After studying all the nuances of architecture (for example, cyclic data flow connection is impossible), the following algorithm was used:

1. Collect all graph leaves.
2. Find all leaves without outcoming connections and add them to the first groups of nodes.
3. Traverse the first group and find all nodes that point to the nodes from this group. Add them to the second group.
4. Repeat 2. and 3. steps while the last created group has any incoming connections.
5. For every node in each group, recreate the full path to the root, ignoring other leaves. After this step, we have groups of branchless graphs.
6. Merge all the graphs in each group among themselves.
7. Reverse groups, so that the connections go from left to right.
8. Final logical data representation is done.

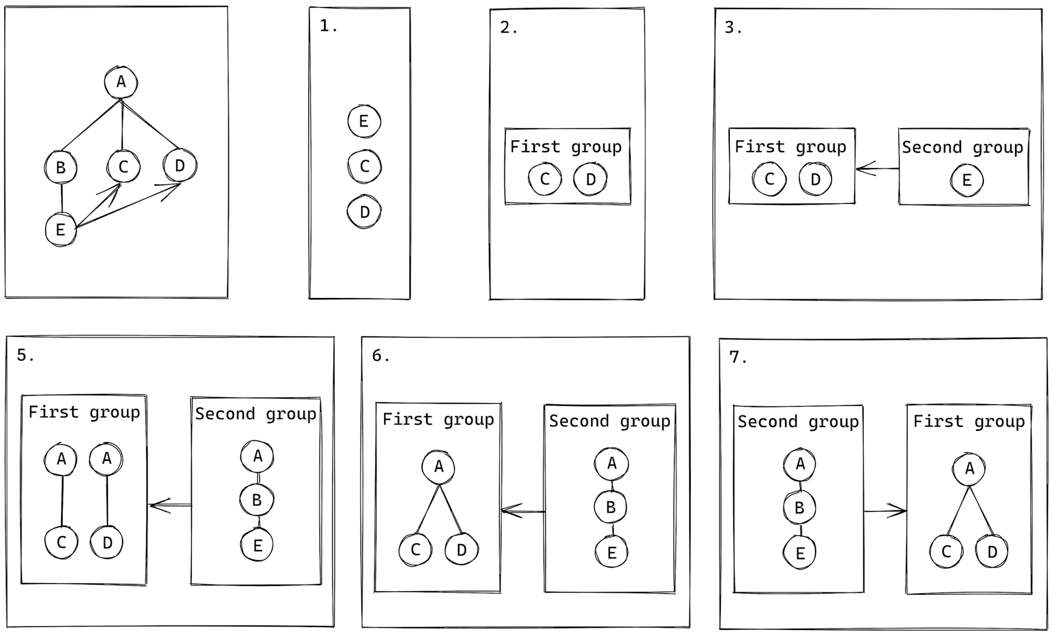
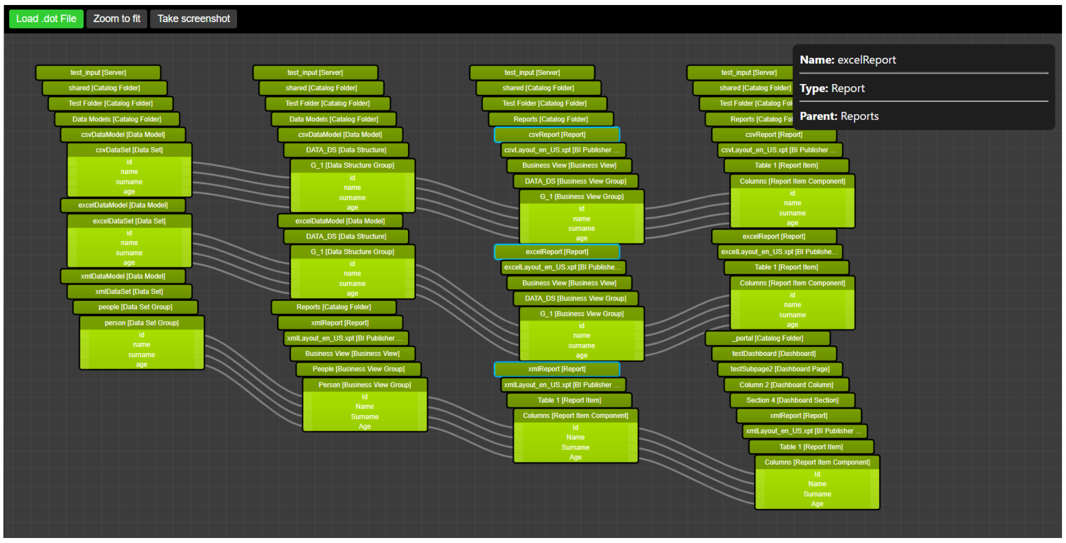


Figure 6.4 How SubGraphBuilder works

6.4 Final result

Figure 6.5 is a view of Dev-Viewer with simple input example:



Name: excelReport
Type: Report
Parent: Reports

Figure 6.5 Dev-Viewer prototype

In addition to the main display task, the following functionality was added:

- Leaves grouping - all neighbor leaves are grouped under their parent node, to combine them visually and thus simplify the perception. That means that such an ancestor node contains its name and all child leaves - for example, 'csvDataSet' node.
- Zoom to fit - the button that performs zooming action that places all (or selected) elements in the visible part of the working area. This action is automatically called after each file upload, to show centered and sized visualization.
- Take screenshot - the button that allows users to download visualization in the form of png image. A snapshot is taken only of the working part of the window without a toolbar.
- Information panel - that is placed in the top right corner of the example is designed to show the developer all the available information. For instance, in case if the name of a node is cropped because of length. The type attribute is not displayed in the node at all (so as not to overload the visualization) and can be found in this panel only. Also, it helps to find the name of a parent node in case of a huge graph.
- Level selector - background feature that highlights all nodes at the same level as the node that was selected.
- Search - it was not done as a separate feature, because it is already well implemented in any browser. Page structure makes it easy to use standard search through a keyboard shortcut "ctrl + f".

7 Usability testing

To test the usability of a newly created tool, Dev-Viewer was sent to two domain experts from Manta for a test. There were multiple rounds of testing on different stages of implementation, thanks to timely expert reviews, development was headed in the right direction. This allows saying that was applied incremental approach for user interface development, which greatly influenced the requirements and the final result for the better

In the meanwhile, it was tested while developing new functionality for the MicroStrategy data flow parser, but to be objective, only the results of tests conducted by other experts will be given. Testers praised successful solutions and pointed out flaws and bugs. Some of the suggestions were already implemented, some of them will need to be done in the future, because of the complexity and prototype status of the tools.

Here are some citations where experts compare Dev-Viewer to the previous solution. [Original comment can be found in attachments as CommentDevViewerPK.odt]

Petr Kosvanec: "Where Graphviz failed in visualizing relations between a parent and a child node, Dev-Viewer shines and provides a detailed overview of the whole hierarchy. Where Graphviz produces only png images, Dev-Viewer is browser-based and allows the user to filter for specific nodes using ctrl+f."

Yauheniy Buldyk: "Compared to Graphviz, the Dev-Viewer shows excellent readability and interactivity. It is possible to search, and there is a native image enlargement, which helps a lot."

Of course, there are not only positive things but also problems. The following things have been highlighted as flaws:

- Graphviz shows edge type, which is missing in Dev-Viewer.
- The pre-filtering of visualized nodes is needed.
- It would be great to add a node full path to the information panel.

But overall, the project was described as useful and convenient:

"I reviewed the developer visualizer, which is the subject of this thesis and it is evident that it aims to do just that, and even in its current form, it can prove much stronger and suitable a tool for our purposes than Graphviz."

"There is still some work to do, namely pre-filtering of visualized nodes, codebase integration, and a few less important issues, but in MANTA, we believe Dev-Viewer will soon replace Graphviz in our company."

8 Conclusion

The main goal of this work was to implement a data flow parser for the MicroStrategy reporting system and using the gained experience, create a functional prototype of a tool that aims to replace outdated technology. A lot of work was done to identify requirements with domain experts and find a suitable solution, with a correct user interface. It was a really productive process.

The created tool is not perfect, and there is still a lot of work for the future. This is just a prototype at this moment, but it already shows great results in real-life usage. There are big plans for further development and improvement. In the foreseeable future, the Dev-Viewer will completely replace the Graphviz in Manta.

For me, it was a very positive experience in creating this type of software, since I first encountered the development of interfaces and web applications in real-life. However, I am very pleased with the result and intend to continue to develop in this direction.

Bibliography

[1] GROW. *WHY IS DATA IMPORTANT FOR YOUR BUSINESS?* 2020-05-09. URL: https://www.grow.com/blog/data-important-business (visited on 05/22/2020).

[2] Sunny Dhami. *What Is Business Intelligence and Why Does It Matter to Enterprises.* 2019-12-20. URL: https://www.ringcentral.co.uk/blog/business-intelligence/ (visited on 05/22/2020).

[3] Aayushi Johari. *What Is Java? A Beginner's Guide to Java and Its Evolution.* © 2002–2019. URL: https://www.edureka.co/blog/what-is-java/ (visited on 05/22/2020).

[4] The Apache Software Foundation. *Apache Maven Project.* 2020-05-07. URL: http://maven.apache.org/ (visited on 05/22/2020).

[5] Yash$_M$*aheshwari. What is JavaDoc tool and how to use it?* URL: https://www.geeksforgeeks.org/what-is-javadoc-tool-and-how-to-use-it/ (visited on 05/22/2020).

[6] tutorialspoint. *TypeScript - Overview.* © 2020. URL: https://www.tutorialspoint.com/typescript/typescript_overview.htm (visited on 05/22/2020).

[7] MicroStrategy. *The MicroStrategy platform.* © 2017. URL: https://www2.microstrategy.com/producthelp/10.4/ProjectDesignGuide/WebHelp/Lang_1033/Content/ProjectDesign/The_MicroStrategy_platform.htm (visited on 05/22/2020).

[8] MicroStrategy. *MicroStrategy REST.* URL: https://demo.microstrategy.com/MicroStrategyLibrary/api-docs/index.html#/ (visited on 05/22/2020).

[9] MicroStrategy. *Creating a report.* URL: https://doc-archives.microstrategy.com/producthelp/10.1/WebUser/WebHelp/Lang_1033/Creating_a_new_report.htm (visited on 05/22/2020).

[10] MicroStrategy. *Designing and creating documents.* © 2016. URL: https://doc-archives.microstrategy.com/producthelp/10.7/ReportDesigner/WebHelp/Lang_1033/Content/ReportDesigner/designing_documents.htm (visited on 05/22/2020).

[11] MicroStrategy. *About Dossiers.* © 2019. URL: https://doc-archives.microstrategy.com/producthelp/10.10/WebUser/WebHelp/Lang_1033/Content/about_analyses.htm (visited on 05/22/2020).