

Jayvee Data Wrangler

MASTER'S THESIS

Elias Pfann

Submitted on 3 June 2024



Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Georg Schwarz M. Sc.
Prof.Dr. Dirk Riehle, M.B.A.



Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 3 June 2024

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

Erlangen, 3 June 2024

Abstract

This thesis aims to explore and explain the creation of an open source software (Jayvee Data Wrangler) that ought to perform data wrangling tasks to help users with importing their data (CSV) into a database, as well as its exploration and editing in a semi-automated way. The software is based on the DSL Jayvee. The Jayvee Data Wrangler is designed to be an easy-to-use software that also aims to simplify the process of generating data pipelines and make the features of ETL pipelines accessible to non-programmers. The software guides users through the process of importing and modifying data, automatically recognizing essential metadata and providing filtering and modification options for the imported data. It generates valid Jayvee code to allow further modification by more experienced users. This thesis also includes an evaluation of the Jayvee Data Wrangler, a comparison with other data wrangling tools, and suggestions for extending the Jayvee Data Wrangler. The source code of the Jayvee Data Wrangler is available at the following URL: <https://github.com/kreisligaspieler/Jayvee-Data-Wrangler>.

Contents

1	Introduction	1
2	Related Work	3
2.1	Data Wrangling and ETL-Pipelines	3
2.2	Related Tools	4
2.2.1	Python Pandas	4
2.2.2	Microsoft Data Wrangler Extension for Visual Studio Code	4
2.2.3	Apache Spark	5
2.2.4	Amazon Sagemaker Data Wrangler	5
2.2.5	Alteryx Analytics Cloud Platform	6
2.2.6	OpenRefine	6
3	Fundamentals	9
3.1	Context: Jayvee	9
3.2	CSV	11
4	Requirements	13
4.1	Requirements Engineering	13
4.2	Functional Requirements	14
4.3	Non-functional Requirements	16
5	Architecture and Design	19
5.1	Architecture of Different Components	19
5.1.1	User Interface	19
5.1.2	Backend	25
5.2	Interaction Between Components	26
6	Details of the Implementation	31
6.1	Components and Structure of the Project	31
6.2	Importing CSV files	35
6.2.1	Generating and Modifying the UI	35
6.2.2	Analyzing CSV files	37

6.3	Modifying data	43
6.4	License Considerations	47
7	Evaluation	49
7.1	Requirements Evaluation	49
7.2	Examples of Testing	50
7.3	Software Classification	51
8	Conclusion and Outlook	55
	Appendices	57
A	Screenshots of the Software	59
B	Interaction Between Components	64
C	Components and Algorithms of the Jayvee Data Wrangler	72
	References	77

List of Figures

1.1	An illustration of a data pipeline oriented to Jayvee.	2
2.1	Interface of Microsoft Data Wrangler.	5
2.2	The interface of OpenRefine.	7
3.1	Example of a CSV file.	12
5.1	User notification if an error occurs during the import of a CVS file.	21
5.2	The UI when the user views the data of a project. The user can now execute multiple commands.	21
5.3	Example of statistics available for a column with integer value type.	24
5.4	Interaction between the user and software, and between different parts of the software.	28
5.5	Interaction between the user and software, and between different parts of the software when the user displays the database.	29
5.6	Interaction between user and software when creating a new value type.	30
6.1	Communication between components of the frontend and backend.	32
6.2	The consistent construction of the Jayvee Data Wrangler.	36
6.3	Steps during the process of importing a CSV into a database.	38
A.1	The entry page of the Jayvee Data Wrangler	59
A.2	Screenshot of the Jayvee Data Wrangler showing a displayed warning	60
A.3	Screenshot of the Jayvee Data Wrangler showing a successful import of a CSV into a database.	60
A.4	Screenshot of the Jayvee Data Wrangler showing the creation of a constraint.	61
A.5	Screenshot of the Jayvee Data Wrangler showing the creation of a value type.	61
A.6	Screenshot of the Jayvee Data Wrangler showing the deletion of multiple rows	62

A.7	Screenshot of the Jayvee Data Wrangler showing the applying of a custom value type	63
A.8	Screenshot of the Jayvee Data Wrangler showing the listing of all created constraints and value types within a project.	63
A.9	Screenshot of the Jayvee Data Wrangler showing an example of a list of created projects.	64
B.1	Interaction between the user and Jayvee Data Wrangler while editing a column name.	64
B.2	Interaction between the user and Jayvee Data Wrangler while changing a value type.	65
B.3	Interaction between the user and Jayvee Data Wrangler while creating a new constraint.	66
B.4	Interaction between the user and Jayvee Data Wrangler while deleting a row.	67
B.5	Interaction between the user and Jayvee Data Wrangler while deleting a column.	67
B.6	Interaction between user and Jayvee Data Wrangler while displaying a folder	68
B.7	Interaction between user and Jayvee Data Wrangler while undoing the last action	69
B.8	Interaction between user and Jayvee Data Wrangler while redoing the last action	70
B.9	Interaction between user and Jayvee Data Wrangler to save changes	71

List of Tables

5.1	Requested user input before the data is imported into a SQLite database	20
5.2	Table showing the age and salary of employees.	22
5.3	This table shows the changes that would happen to table 5.2 if the value type of the age column would be changed to integer.	22
5.4	Constraints that are supported by Jayvee (The JValue Project, 2024).	23
6.1	Example showing the change of the value type of two columns. . .	45
7.1	Summary of requirement fulfillment	50
7.2	Features of the Jayvee Data Wrangler in comparison to other software.	53

Listings

3.1	Example of a Jayvee pipeline.	10
6.1	Inter-process communication (IPC)	34
6.2	Creating the window of the application and loading the startpage.	35
6.3	Function to create editable paragraphs used in the Jayvee Data Wrangler	36
6.4	Starting the process of importing a CSV into a database using IPC.	37
6.5	Function to listen on an IPC channel to receive data from other files.	38
6.6	Identify the CSV Delimiter	40
6.7	Function to create the Jayvee Script	42
6.8	Dynamical loading of data into the datatable.	44
6.9	Example of validating values against constraints.	45
6.10	Excerpt from the algorithm that modifies the display of values.	46
C.1	Content of the preload file used to expose functions securely privileged into the renderer process.	72
C.2	Pipeline object to store data that will be inserted into the Jayvee file.	72
C.3	Creation of the project folder within the workspace in the user directory.	73
C.4	Function that loads <i>viewDatabase.html</i> into the main window and hands over the database path and table name.	73
C.5	Creating a datatable to view the database	74
C.6	Adding data to the data table	75

Acronyms

AWS	Amazon Web Services
CSV	Comma-Separated Value
DSL	Domain Specific Language
ETL	Extract, Transform, Load
FR	Functional Requirements
IPC	Inter-Process Communication
NFR	Nonfunctional Requirements
OS	Operating System
UI	User Interface

1 Introduction

In today's technologically advanced world, an enormous amount of data is collected from various different sources. Not only smart devices, but also scientific measuring stations (for example) gain data. Structuring and analyzing large amounts of data, like for training an AI or building an application, has become a big industry. Finding and recognizing connections and correlations in datasets can be very profitable, not only for companies, but also for scientific research.

In order to establish correlations between the data, data engineers must first take on the task of processing this data from different sources, which have different data formats and varying levels of quality. This processing often involves transforming the data into formats that are easily understood by humans, as well as converting it into machine-readable formats, often using programming languages such as Python. This entire process is commonly referred to as *data pipeline*. It requires either programming experience or a significant amount of manual work. Although automated tools can speed up this process and generate evaluations of the data, scientists still struggle with the complexity and high manual effort involved in this process.

Therefore, this thesis analyzes the possibility of assisting users in transforming, structuring, cleaning and exploring collected data in a semi-automated way using specialized software, which not only simplifies the process of creating a data pipeline, but also makes automated data evaluation accessible to non-programmers. For this purpose, the Jayvee Data Wrangler was developed. With this software users can generate data pipelines without any programming being required. The software uses the open-source Domain Specific Language (DSL) Jayvee to define Extract, Transform, Load (ETL) pipelines (refer to figure 1.1) that import data from sources to sinks to obtain the data in a valuable format for further processing. With an unpretentious interface, users are automatically guided through the process of importing data into a database. During this process, the essential metadata (e.g. encoding) is as far as possible automatically detected to eliminate or reduce any effort. Once imported, the data can be filtered and modified to suit the user's needs. To support this process and to get an overview over the data, users are also provided with several descriptive statistics. Under the hood, the Jayvee Data Wrangler uses Typescript to generate a

valid Jayvee file that loads the contents of a Comma-Separated Value (CSV) file into a suitable sink type (e.g., a SQLite file). Users with programming experience can modify and customize the pipeline by editing the Jayvee file.

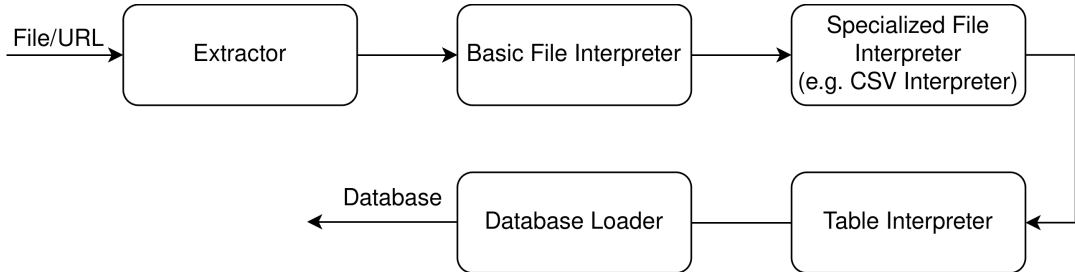


Figure 1.1: An illustration of a data pipeline oriented to Jayvee.

This thesis is organized as follows: A brief overview of the related work and software is given in chapter 2. Fundamentals such as Jayvee and CSV are introduced in chapter 3. Chapter 4 presents the necessary requirements for the successful implementation of the tool. Chapter 5 shows the implemented software architecture, while chapter 6 discusses and illustrates the implementation in detail. Furthermore, in chapter 7 the software is evaluated by testing it with example workflows and comparing it to other tools. Finally, chapter 8 provides a conclusion and an outlook on possible future work.

2 Related Work

There is no data wrangling tool based on Jayvee yet. Therefore, this chapter provides an overview of similar software/libraries and related tools, while Jayvee itself will be introduced in chapter 3. Not only for a better understanding of this chapter, but also in order to gain a better comprehension of the Jayvee Data Wrangler's features, specific concepts and technical terms like ETL and data pipelines are explained at the beginning.

2.1 Data Wrangling and ETL-Pipelines

Data wrangling involves identifying, extracting, cleaning and integrating the data needed for an application. It is sometimes described as converting data from a raw format into a more convenient format for analysis with other tools (Furche et al., 2016). It helps to maintain accuracy by cleaning the data, which is crucial for making accurate conclusions and decisions based on the data. It can also ensure consistency among different data sources by standardizing the format and structure of the data. It furthermore ensures completeness by addressing missing values through data cleaning. Data wrangling also produces higher quality data. At the same time, it can increase efficiency by saving time in dealing with data issues during later stages of the project (Endel and Piringner, 2015).

While data wrangling is a broader concept that encompasses all the activities involved in preparing data for analysis, ETL refers to the process of extracting data from different sources, transforming it into a format that is suitable for analysis and loading it into a target database or data warehouse (Furche et al., 2016).

According to Voleti, 2020 there are some obstacles surrounding data wrangling:

- Time consumption
- Ensuring data compatibility and consistency across various sources
- Lack of direct data access
- Complexity in data mapping

- Need for technical expertise
- Data quality and reproducibility

These obstacles can be addressed by various data wrangling tools. All of these tools improve workflow, save time and reduce complexity while deriving valuable insights from a data source.

2.2 Related Tools

There are many data wrangling tools that perform (semi)automated data import. Several widely-used data wrangling tools that support the import of at least CSV files are reviewed to determine the features they offer. All tools also have in common that they clean and prepare data, and support data exploration.

2.2.1 Python Pandas

Pandas¹ is a popular open-source data manipulation library for Python. It provides data structures and functions for efficiently cleaning, transforming and analyzing data. Programmers cannot only use it to import and filter data, but also to visualize it (NumFOCUS, Inc., 2024). When used with other Python libraries, it can be used to perform many conceivable of data-based tasks.

2.2.2 Microsoft Data Wrangler Extension for Visual Studio Code

Microsoft Data Wrangler is a Visual Studio Code extension. It works with either CSV files, Parquet files² or Pandas DataFrames³. Programmers can also get statistics through its visualization capabilities, which automatically generate a preview of the data called *quick insights* (refer to figure 2.1). It automatically loads files and has various operations for filtering the data like applying mathematical formulas or dropping specific values. To outline a few aspects of it: Users can change the column type, rename the header and export data to a CSV file, Parquet or a Jupyter Notebook⁴. The extension automatically generates Python Pandas code that can also be viewed and exported (Mew, 2023).

¹<https://pandas.pydata.org/>

²<https://parquet.apache.org/>

³<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>

⁴<https://jupyter.org/>

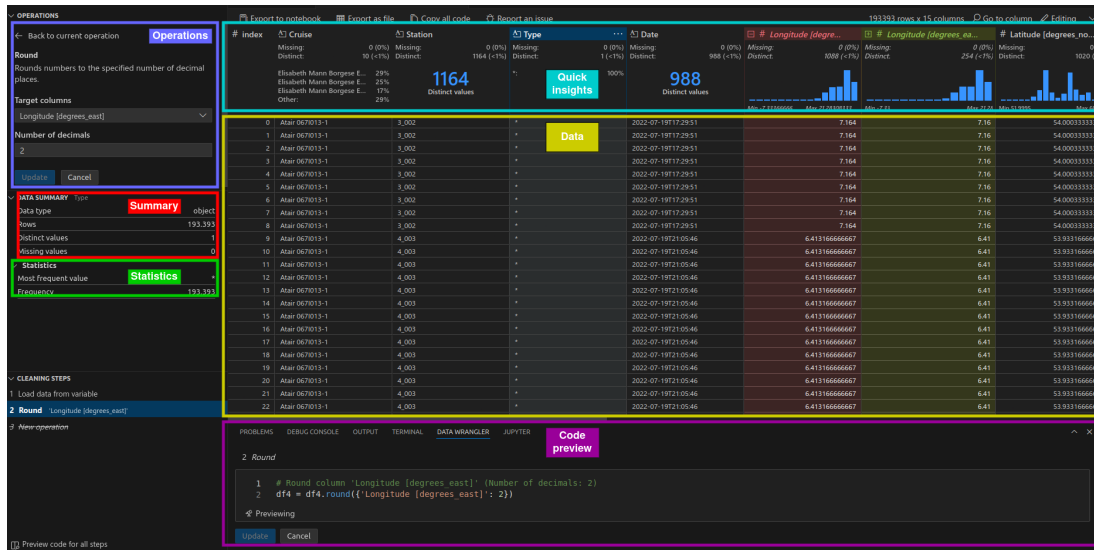


Figure 2.1: Interface of Microsoft Data Wrangler.

2.2.3 Apache Spark

Apache Spark⁵ is an open-source data processing engine that supports data wrangling tasks for big data workloads. It supports batch/streaming data, SQL analytics, Exploratory Data Analysis on large quantities of data and training of machine learning algorithms. Spark provides support of programming languages like Scala, Python, Java, and R (Apache Software Foundation, 2024). The software is designed to run on servers (on top of Apache Hadoop) and clusters of servers. It provides and achieves faster performance by utilizing in-memory caching, as well as optimized execution. With its API and special libraries, such as Spark SQL or a machine learning library, multiple tasks can be performed (Murthy, 2017). While Spark SQL is easy to understand, programming experience is required to take full advantage of Apache Spark.

2.2.4 Amazon Sagemaker Data Wrangler

This software⁶ is integrated in the Amazon Web Services (AWS) ecosystem with little to no coding required (Amazon Web Services Inc., 2024a). With over 300 built-in data transformations, it enables easy and fast data preparation. Data cannot only be imported and cleaned, but also visualized. Workflows can even be automated using autopilots. Users can leverage its integration with other AWS services to explore and analyze data (Amazon Web Services Inc., 2024b).

⁵<https://spark.apache.org/>

⁶<https://aws.amazon.com/de/sagemaker/data-wrangler/>

It is also available as Python library and works with Python Pandas DataFrames (AWS Professional Service, 2024).

2.2.5 Alteryx Analytics Cloud Platform

This software⁷ runs on the Alteryx cloud environment and also supports machine learning, as well as automated and repeatable data pipelines. Using ETL pipelines, users can generate automated analytics without programming required. The platform also supports the creation of statistics and reports, and supports a variety of data sources such as AWS or Google Cloud, basic file formats like CSV, JSON and even raw HTML pages (Alteryx, Inc., 2024b and Alteryx, Inc., 2024c). Generative AI allows users to automatically generate insights and recommended analytics based on the user’s business (Alteryx, Inc., 2024a).

2.2.6 OpenRefine

OpenRefine⁸ was initially released by Google in 2010 and is regularly maintained. It is published under the open source BSD license and can be downloaded for Windows, macOS and Linux. The only requirements are having the supported Java version and a supported browser installed. In addition to standard data wrangling operations, it also supports clustering to find inconsistencies, reconciliation to match a dataset against another, infinite undo and redo, — and because it runs local — it protects the privacy of the data. OpenRefine is very easy to install and use without writing a single line of code. Users have the option to filter, explore and apply modifications to the data. The software is very good at recognizing different file formats. Transforming and editing data as well as exporting it (e.g. to CSV or SQL) is straight forward (OpenRefine, 2024 and OpenRefine developers, 2024). The interface of OpenRefine is shown in figure 2.2 below.

⁷<https://www.alteryx.com/products/alteryx-platform>

⁸<https://openrefine.org/>

2. Related Work

The screenshot displays the OpenRefine web interface for a CSV file named 'MO H CHLA 2024.csv'. The main area shows a table with 1059 rows. The columns are: ID, Status, Type, yyyy-mm-ddThh:mm:ss.sss, Longitude (degrees_west), Latitude (degrees_north), LOCAL_CD_ID, CDMO_code, Bnd. Depth (m), Depth (m), QV, CHEA_SBE31PLUS_SENSE_DEV_NOT (µg/l), and QV2. The table contains data for various sampling events, with most having a 'Status' of 'A' and a 'Type' of 'A'. The 'LOCAL_CD_ID' and 'CDMO_code' columns are mostly empty or contain 'BSH'. The 'Depth' and 'QV' columns contain numerical values. The 'CHEA_SBE31PLUS_SENSE_DEV_NOT' column contains values like 24.5384, 24.2576, 24.5245, etc. The 'QV2' column contains values like 1, 1, 1, etc. A sidebar on the left titled 'Using facets and filters' provides instructions on how to use the interface.

ID	Status	Type	yyyy-mm-ddThh:mm:ss.sss	Longitude (degrees_west)	Latitude (degrees_north)	LOCAL_CD_ID	CDMO_code	Bnd. Depth (m)	Depth (m)	QV	CHEA_SBE31PLUS_SENSE_DEV_NOT (µg/l)	QV2
1	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	15	1	24.5384	1
2	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	14	1	24.2576	1
3	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	13	1	24.5245	1
4	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	12	1	22.6587	1
5	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	11	1	22.7961	1
6	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	10	1	22.4151	1
7	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	9	1	21.931	1
8	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	8	1	18.2135	1
9	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	7	1	15.9429	1
10	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	2	1	6.9552	1
11	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	3	1	7.4429	1
12	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	6	1	14.0961	1
13	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	5	1	9.8686	1
14	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	23	1	36.2852	1
15	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	22	1	36.524	1
16	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	21	1	33.1574	1
17	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	20	1	29.226	1
18	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	4	1	7.6537	1
19	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	19	1	28.0261	1
20	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	18	1	27.7598	1
21	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	17	1	26.6801	1
22	A	A	2024-04-23T06:57.21	8.109	54.0001666666667		BSH	25	16	1	24.6071	1
23	A	A	2024-04-23T11:12.58	7.9675	54.0486666666667		BSH	30	17	1	25.5364	1
24	A	A	2024-04-23T11:12.58	7.9675	54.0486666666667		BSH	30	26	1	29.2872	1
25	A	A	2024-04-23T11:12.58	7.9675	54.0486666666667		BSH	30	27	1	34.371	1

Figure 2.2: The interface of OpenRefine.

2. Related Work

3 Fundamentals

This section elucidates the two key concepts/parts of software that are needed to be understood in order to comprehend the development and functionality of the Jayvee Data Wrangler. First, the Jayvee DSL language is illustrated to understand the creation and use of Jayvee files for the process of importing CSV files into a database. It is followed by the exposition of the structure of CSV files to understand the complexity of properly importing the contents of any CSV file.

3.1 Context: Jayvee

As described in the introduction, the Jayvee Data wrangler builds upon the open-source DSL Jayvee which describes ETL pipelines from sources to sinks to obtain the data in a suitable format for further processing. The interpreter allows executing such data pipelines on a local machine. Jayvee can be used to prepare, filter and clean data for applications such as machine learning or data analysis. Because it is open source, users (even non-programmers) are invited to collaborate so that the developers can improve and extend the software.

Jayvee is structured around four basic principles that are easy to understand:

- Pipelines: Sequences of different computing steps (blocks).
- Blocks: A processing step within a pipeline that can have standard input and standard output.
- Value type: Determine the data type of the processed data. There are built-in value types and primitive value types (user-defined value types).
- Transforms: Transformations are used to convert data from one value type to another value type. (The JValue Project, 2024)

In order to facilitate a deeper understanding of the Jayvee pipelines created by the Jayvee Data Wrangler, an example is provided to explain fundamental underlying concepts:

3. Fundamentals

```
1  /* The pipeline consists of different blocks chained through pipes ("->"),
2  while the output of one pipeline is used as input for another. It usually
3  starts with a definition of the order of the blocks leading to an overview
4  of the pipeline. */
5  pipeline TrainstopsPipeline {
6      TrainstopsExtractor -> TrainstopsTextFileInterpreter
7      ->TrainstopsCSVInterpreter->TrainstopsTableInterpreter
8      ->TrainstopsLoader;
9
10     /* Through the use of the oftype keyword, blocks are able to generate
11     instances of a particular blocktype. The HttpExtractor block specifies
12     a URL where the file shall be downloaded from. It produces a binary
13     file as output, which is passed to a TextFileInterpreter. */
14     block TrainstopsExtractor oftype HttpExtractor{
15         url:"https://download-data.deutschebahn.com/static/datasets/
16             haltestellen/D_Bahnhof_2020_alle.CSV";
17     }
18     // This block defines how to interpret the file.
19     block TrainstopsTextFileInterpreter oftype TextFileInterpreter {
20         encoding="utf-8";
21     }
22
23     /* With the CSVInterpreter block, the file will be interpreted as a
24     CSV file with a specified enclosing and delimiter. */
25     block TrainstopsCSVInterpreter oftype CSVInterpreter {
26         enclosing: "'";
27         delimiter: ";";
28     }
29
30     // With TableInterpreter the datatype for each column is defined.
31     block TrainstopsTableInterpreter oftype TableInterpreter {
32         header: true;
33         columns: ["EVA NR" oftype integer,
34                 "DS100" oftype text,"IFOPT" oftype text,
35                 "NAME" oftype text,"Verkehr" oftype text,
36                 "Laenge" oftype text,"Breite" oftype text,
37                 "Betreiber_Name" oftype text,
38                 "Betreiber_Nr" oftype integer,
39                 "Status" oftype text];
40     }
41
42     /* Finally, the table is loaded into a sink, in this case a SQLite
43     database, using the previously defined structural information. */
44     TrainstopsLoader Loader oftype SQLiteLoader {
45         table: "trainstops";
46         file: "./trainstops.sqlite";
47     }
```

Listing 3.1: Example of a Jayvee pipeline.

The example in listing 3.1 describes a simple Jayvee pipeline from a CSV file on the web to a SQLite file sink. The code is an example of an ETL pipeline (explained in figure 1.1) and has been annotated with comments for easier understanding. There are many other options available including deleting rows, filtering data, renaming column names or defining custom value types. Users are able to even work with zip files or import data from XLSX files (The JValue Project, 2024). Due to the lucid syntax, other concepts are also relatively straightforward to read and understand. More information about them can be found in the of-

ficial docs¹, which also provide further precisely annotated examples. Following the instructions, Jayvee can easily be installed and updated with this documentation. From a developer's point of view, even error messages are precise and easy to understand, making development uncomplicated. Of course, there are many alternatives on the market like Python Pandas, to name just one. Instead, Jayvee was chosen for the development of the data wrangler because of its simple and clean structure, including all the presented features. As it is under active development, additional useful features are continuously being added, which can be incorporated into the Jayvee Data Wrangler.

3.2 CSV

The Jayvee Data Wrangler is designed to import CSV files into databases. CSV files can be compared to spreadsheets and tables (e.g. Excel spreadsheets) where the entries are organized in rows and columns, but they also have some other properties. According to Shafranovich, 2005 there is no formal specification for CSV files in existence, only a format that most CSV files follow. Looking at the RFC 4180 standard (as defined by Shafranovich) helps to identify the obstacles that may arise when developing the Jayvee Data Wrangler.

In his work, Shafranovich, 2005 identifies several common traits of CSV files (pages 1-2). They are annotated with comments to identify challenges and present solutions to any problems that may arise during the import process:

1. A line break delimits each record located on a separate line. Therefore, to analyze the file, it can be read line by line.
2. The last record in the file may or may not have an ending line break (newline), which is not a problem, because even if the entire file is read for analysis, the last record will be recognized by the end of the file.
3. The first line may contain an optional header line having the same format as normal record lines. The header functions as a column name and should contain the same number of fields as the records in the files. If there is no header, the Jayvee Data Wrangler assigns a custom header based on the number of entries in each row.
4. The header and the records are separated by commas. The number of fields in two separate lines should not differ. The last field in a record might not have a comma. Unlike Shafranovich, the Jayvee Data Wrangler also recognizes semicolons as delimiters, because they are also used to separate fields within the records.
5. Each field may be enclosed in double quotation marks. If the enclosing of a

¹<https://jvalue.github.io/jayvee/docs/user/intro/>

3. Fundamentals

field is double quotes, then double quotes may not appear within the fields. Sometimes fields are also enclosed with single quotes, so the Jayvee Data Wrangler covers that.

6. Fields are usually enclosed in double quotation marks if they contain line breaks, double quotation marks or commas. If they are not enclosed in either single or double quotes, commas may be recognized as the entry delimiters.
7. Any double-quotes within an entry must be escaped with a preceding double quote if the enclosing is double-quoted.

Observations and tests during the development of the Jayvee Data Wrangler showed that multiple lines containing comments or empty lines may appear before the header. These comments are usually preceded by a special character or a sequence of characters (e.g. # or //) to identify them. In addition, the entries of a CSV file can be delimited not only by commas, but also by other characters like semicolons, spaces, or tabulators. During the testing and research, it was also observed that some CSV files do not have the file extension .csv, but are rather saved as .txt files. This allows users to view the file in a web browser. As long as these files follow the previously presented rules by Shafranovich (with the annotations), the Jayvee Data Wrangler will also support their import. In order to demonstrate the previously mentioned terms, an example of how a typical CSV could look like is given in figure 3.1. The illustration showcases the terms enclosing, header, delimiter and content by utilizing a sample record.

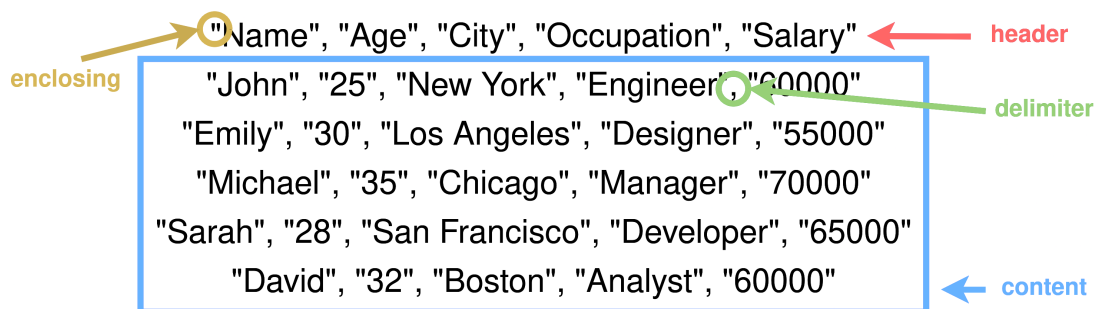


Figure 3.1: Example of a CSV file.

4 Requirements

This chapter describes the necessary requirements for implementing the Jayvee Data Wrangler. The requirements are categorized into Functional Requirements (FR), listed in section 4.2, and Nonfunctional Requirements (NFR), listed in section 4.3 associated with necessary sub-requirements. The syntax of the requirements is based on the requirement templates by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (Rupp and SOPHISTen, 2020). Before talking about those requirements, some thoughts and conclusions for the implementation based on other previously presented data wrangling software/-programming libraries are explained and some primary considerations based on potential groups of users are made (section 4.1) to determine all the necessary aspects the Jayvee Data Wrangler should have. It is also important to note that Jayvee Data Wrangler is a research prototype and not a commercial software. Therefore, its development focuses on certain basic concepts that a data wrangler should have.

4.1 Requirements Engineering

To define the necessary requirements for the Jayvee Data Wrangler, other tools should be considered as well as potential users identified.

The research in the previous chapter concluded that the Jayvee Data Wrangler must have basic data wrangling capabilities like identifying, extracting, cleaning and integrating data. Because it is built on top of the Jayvee ETL language, it automates tasks that a user — building an ETL pipeline with Jayvee — would perform. These tasks are deciding how the data that is imported is formatted, if it has a header, and how it will be stored in a database. In case of importing CSVs (that are best supported by Jayvee) it would include extracting encoding, delimiter and enclosing. It also shall detect and remove comments. Since it is a prototype and not an enterprise software, the Jayvee Data Wrangler does not need to be able to create complex evaluations and statistics, but be able to import, structure and filter the data that will be stored for further analysis.

As the software should have a wide range of applications, there are at least

two potential groups of users: Users with no programming experience and users with programming knowledge. Non-programmers, such as university researchers, should be able to use the Jayvee Data Wrangler without any knowledge of Jayvee and without having to write a single line of code. They may use the software, extract data from a CSV file into a database for further processing and may also want to filter the data to remove unwanted entries. To avoid obstacles, they need an application that's easy to understand and also provides them with clean documentation that explains when and why they have to insert some data or parameters.

Users with some programming knowledge may be interested in modifying more than only a few basic parameters. They should be able not only to import a CSV file, but also to view and modify the underlying Jayvee-script. By saving the script permanently, they could modify it after importing the data, extend it with further functionality or even use it for automated importing if the data source changes.

With both groups of users in mind, the Jayvee Data Wrangler will be, as mentioned in the introduction of this thesis, a software that on the one hand, given a direct URL to a CSV file can import its content into a local database and on the other hand, a tool that automatically generates modifiable Jayvee scripts. Based on these considerations, the requirements of this concrete software are specified.

4.2 Functional Requirements

Functional requirements contain a description of the functionality that the system is intended to provide or a feature that the system should have. They are presented based on the FunctionalMASTeR template created by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (Rupp and SOPHISTen, 2020). This template is a detailed and structured approach to capturing functional requirements in software development projects in a structured and natural language. It identifies the object of consideration, which in this case is the system that is to be constructed. By using the terms "shall" (legally binding), "should" (strongly recommended) and "will" (used in future) the importance of the described functionality concerning the object is determined. Using logical and temporal conditions, FRs are divided into 3 categories using logical expressions ("If"), conditions based on events ("As soon as"), and conditions based on time periods ("As long as").

- FR-1** The Jayvee Data Wrangler shall provide to the user the ability to import CSV Files from the internet via a direct file URL (semi) automatically into a database.
- FR-1.1** The user shall be prompted to insert a URL which directly leads to a CSV file.
 - FR-1.2** In order to process the file, it shall be checked if the file is a valid CSV file.
 - FR-1.3** As far as possible, metadata shall be recognized automatically.
 - FR-1.4** If some metadata needed to start the import of the CSV file is not recognized automatically, the user shall be asked to provide it.
 - FR-1.5** The user shall be able to edit the metadata encoding, delimiter and enclosing before importing the CSV into a database.
 - FR-1.6** The file shall only be processed if all necessary metadata was recognized atomically or given by the user.
 - FR-1.7** The database shall be stored permanently.
- FR-2** The user shall be able to modify and filter the data after importing it.
- FR-2.1** The user shall be able to delete rows and columns.
 - FR-2.2** The user shall be able to rename column descriptions.
 - FR-2.3** The user shall be able to modify the datatype of a column.
 - FR-2.4** The user shall be able to generate custom data types.
 - FR-2.5** Modifying the database shall follow the concepts of Jayvee. ¹
 - FR-2.6** If the content of a database was modified inside the Jayvee Data Wrangler, the user shall be able to save the modified database.
- FR-3** The Jayvee Data Wrangler shall generate valid Jayvee-Scripts.
- FR-3.1** The Jayvee Data Wrangler shall generate a single Jayvee-Script for every CSV file.
 - FR-3.2** As soon as the script was generated successfully, it shall be executed to generate a database and import the data from the CSV into the database.
 - FR-3.3** If a database generated by the Jayvee Data Wrangler was updated using the Jayvee Data Wrangler, the Jayvee-Script shall be updated.

¹<https://jvalue.github.io/jayvee/>

FR-3.4 The Jayvee-Scripts shall be editable and executable without the Jayvee Data Wrangler.

FR-4 The Jayvee Data Wrangler shall be designed in a way that it's easy to use.

FR-4.1 The Jayvee Data Wrangler shall be designed in a way that it provides a clear User Interface (UI).

FR-4.2 The Jayvee Data Wrangler shall have a short manual which is accessible from inside the software.

FR-5 The application should provide statistics about the data.

FR-5.1 The Jayvee Data Wrangler should include statistics about the data within each column.

FR-5.2 The statistics should adapt if the user changes the content of the database inside the Jayvee Data Wrangler.

FR-6 The Jayvee Data Wrangler shall deal with invalid input.

FR-6.1 The software shall only accept a limited set of characters for every input.

FR-6.2 If a character is invalid, the software should notify the user that invalid characters were inserted and not proceed.

4.3 Non-functional Requirements

Non-functional requirements are requirements that are not just focusing on the functionality of the system, but also on aspects related to its development or delivery. If they describe properties of the new feature, they are formulated based on the PropertyMASTeR template; if they are demanded by the operating environment, they are formulated based on the EnvironmentMASTeR template. Both templates are designed by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (Rupp and SOPHISTen, 2020). They both use the same terms as the FunctionalMASTeR to describe the properties of respectively environmental influence to the system.

NFR-1 The Jayvee Data Wrangler should be a standalone application.

NFR-1.1 As long as the user installs all dependencies, the Jayvee Data Wrangler shall be executable using the source code.

NFR-1.2 The Jayvee Data Wrangler should be an Electron.js application which can be executed on Windows, Linux and macOS.

NFR-2 The Jayvee Data Wrangler should be designed in a way that it provides compatibility for possible feature extensions in the future.

NFR-2.1 The source code shall not depend on proprietary libraries or software.

NFR-2.2 The Jayvee Data Wrangler will be published under a license that allows others reusing and extending the software.

NFR-4 The user should have the option to report bugs, suggest features and contribute to the Jayvee Data Wrangler.

NFR-4.1 The Jayvee Data Wrangler will be published on GitHub so that users can report bugs and features there.

NFR-4.2 The Jayvee Data Wrangler will be open source so that other developers can use the software and contribute to the project.

NFR-4.3 The source code should be documented extensively so that other developers can contribute to the development.

4. Requirements

5 Architecture and Design

This chapter provides a detailed description of the architecture of the Jayvee Data Wrangler and how the requirements are implemented (including screenshots of the software). Additional screenshots are included in the section A of the appendix. First the function and behavior of each component is explained, starting with the frontend/user interface (section 5.1.1) and then moving on to the backend (section 5.1.2), while the detailed implementation including source code is explained in chapter 6. Finally, the interaction between the user and the software, as well as the interaction between the individual components of the software, are shown (section 5.2). It's important to note that any mention of Jayvee in this chapter and the following chapters refers specifically to Jayvee version 0.4.0.

5.1 Architecture of Different Components

The Jayvee Data Wrangler is split into two main components: *UI/frontend* and *backend*.

5.1.1 User Interface

The frontend requests user input and displays the results of any data processing (e.g. determine delimiter). The UI also pre-validates the input by restricting each input to a limited set of characters — and if necessary — also a limited number of characters. If characters are not validated in the UI, the backend verifies them. For example, the software sends the URL to the backend without any character restrictions, because the backend code checks if a URL is valid. The restrictions in the frontend are necessary, so that the user cannot carry out unwanted actions, such as overwriting or creating a folder outside the workspace, or inserting characters that the software may not be able to display. Valid input is sent to the backend for further validation and processing. In the UI the data shown in table 5.1 is requested before importing the contents of a CSV into a SQLite database. Encoding, delimiter and enclosing have to be inserted/selected if they are not automatically recognized by the backend, but can also be edited

if they were detected. The encoding is limited to the encodings supported by Jayvee (at the time of delivery of the thesis): utf8, ibm866, latin2, latin3, latin4, cyrillic, arabic, greek, hebrew, logical, latin6, utf-16. To help the user in deciding which values to select, a preview of the first line of data is displayed.

Data	Purpose	Restriction	Mandatory?
Folder name	Create a unique folder within the workspace for storing the CSV, database and Jayvee script	A-Za-z0-9_() and length: 255 (max folder length in windows)	✓
CSV URL	URL that points to the CSV file		✓
Encoding	Encoding of the CSV file	Only encodings that are supported by Jayvee	Only if not recognized automatically
Delimiter	Delimiter that separates the CSV entries	1 character	Only if not recognized automatically
Enclosing	Character that encloses the entries	1 character	Only if not recognized automatically

Table 5.1: Requested user input before the data is imported into a SQLite database

After the data is successfully imported, some metadata — like the value type of a column — is stored in an additional database (metadata database) for loading it when the project is reopened. The user can now click a button to view the contents of the database. If an error occurs, e.g. the encoding is not specified, the user gets notified and has to change values as it is shown in figure 5.1. The first line of data in the file, which helps to identify errors in the metadata, is also shown there.

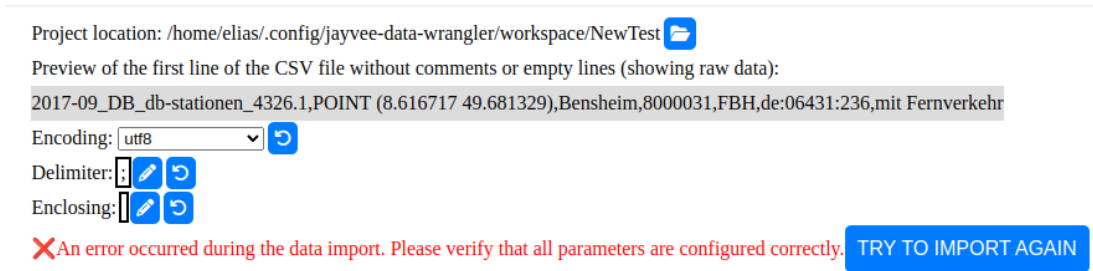


Figure 5.1: User notification if an error occurs during the import of a CSV file.

While viewing the data, users can modify the header names, delete columns or rows and also edit the value type of a column. The UI can be observed in figure 5.2, with detailed screenshots of possible editing actions provided in the appendix A. The Jayvee Data Wrangler even allows the creation of a custom value type with constraints. This custom value type can then be applied to one or more columns. Again, each time the user has to insert a character, there are restrictions to the characters (only A-Z, a-z and 0-9 are allowed) which are, of course, also communicated to the user. Until the user saves the changes, they are reversible via undo and redo buttons. Because this software uses Jayvee to import the contents of a CSV into a database, all inputs and all changes made by the user to the data will follow the specifications of Jayvee ¹.

Jayvee Data Wrangler

Search:

CREATE VALUETYPE CREATE CONSTRAINT VIEW VALUETYPES AND CONSTRAINTS If you edit anything, you can only use following characters: A-Z, a-z and 0-9

	FID	geom	Station	Haltestellennummer	Betriebsstelle	Haltestellenschlüssel	Verkehr
Values	text	text	text	Integer	text	text	text
Actions							
1	2017-09_DB_db-stationen_4326.1	POINT (8.616717 49.681329)	Bensheim	8000031	FBH	de:06431:236	mit Fernverkehr
2	2017-09_DB_db-stationen_4326.2	POINT (9.338469 49.441157)	Seckach	8000042	RSE	de:08225:0203	nur Regionalverkehr
3	2017-09_DB_db-stationen_4326.3	POINT (8.167962 49.564384)	Grünstadt	8000137	RGR	de:07332:1009	nur Regionalverkehr
4	2017-09_DB_db-stationen_4326.4	POINT (8.675442 49.403567)	Heidelberg Hbf	8000156	RH	de:08221:1160	mit Fernverkehr
5	2017-09_DB_db-stationen_4326.5	POINT (8.126204 49.197741)	Landau(Pfalz)Hbf	8000216	RLA	de:07313:8304	nur Regionalverkehr
6	2017-09_DB_db-stationen_4326.6	POINT (8.433402 49.477987)	Ludwigshafen(Rh)Hbf	8000236	"RL	RL T"	de:07314:2080
7	2017-09_DB_db-stationen_4326.7	POINT (8.468921 49.479354)	Mannheim Hbf	8000244	RM	de:08222:2417	mit Fernverkehr
8	2017-09_DB_db-stationen_4326.8	POINT (8.273146 49.045678)	Wörth(Rhein)	8000254	RWRT	de:07334:1731	nur Regionalverkehr
9	2017-09_DB_db-stationen_4326.9	POINT (8.356448 49.63494)	Worms Hbf	8000257	FWOR	de:07319:4415	mit Fernverkehr
10	2017-09_DB_db-stationen_4326.10	POINT (9.113093 49.348484)	Mosbach-Neckarelz	8000264	RNZ	de:08225:9180	nur Regionalverkehr

Showing 1 to 10 of 203 entries

1 2 3 4 5 ... 21

Figure 5.2: The UI when the user views the data of a project. The user can now execute multiple commands.

¹<https://jvalue.github.io/jayvee/docs/user/intro/>

Deleting rows and columns modifies them visually by crossing out each cell and sending the changes to the backend. Changing the value type needs a little more explanation. There are four built-in value types in Jayvee: text, integer, decimal and boolean. To meet Jayvee's specifications, each entry is checked to see if it meets the requirements of the new value type. For example, changing a column from double to integer requires checking each entry to see if the value is an integer (which only applies for a subset of decimals). Changing from text to integer would delete all entries in that column that are not integers, even empty entries. Jayvee only allows empty entries in rows that have the value type text. So if a column's value type is changed to other than text, removing every invalid entry results in the corresponding row being deleted. This is illustrated in the following example:

	name	age	income
ValueType	text	text	integer
1.	Thomas	39	50000
2.	James	40	48000
3.	Sarah	35	55000
4.	Lisa		1000000
5.	Carl	42	69000

Table 5.2: Table showing the age and salary of employees.

	name	age	income
ValueType	text	integer	integer
1.	Thomas	39	50000
2.	James	40	48000
3.	Sarah	35	55000
4.	Lisa		1000000
5.	Carl	42	69000

Table 5.3: This table shows the changes that would happen to table 5.2 if the value type of the age column would be changed to integer.

By changing the value type of the column *age* from text to integer, the fourth row of data is removed, as demonstrated in this example.

Another feature of the Jayvee language (and also of the Jayvee Data Wrangler) is the creation of custom value types. They consist of built-in value types and one or more constraints. These constraints limit the contents of entries to a set of specified values. Hence, creating custom value types requires creating a suitable

constraint first. The user can choose out of five different types of constraints listed in table 5.4 which are compatible only with certain value types.

Constraint	Base value type	Description
AllowlistConstraint	text	Limits the values to a defined set of allowed values. Only values in the list are valid.
DenylistConstraint	text	Defines a set of forbidden values. All values in the list are considered invalid.
LengthConstraint	text	Limits the length of a string with an upper and/or lower boundary. Only values with a length within the given range are valid.
RangeConstraint	decimal, integer	Limits the range of a number value with an upper and/or lower boundary, which can be inclusive or exclusive. Only values within the given range are considered valid.
RegexConstraint	text	Limits the values complying with a regex. Only values that comply with the regex are considered valid.

Table 5.4: Constraints that are supported by Jayvee (The JValue Project, 2024).

The Jayvee Data Wrangler also guides the user through the process of creating a value type. After naming the value type and selecting the base value type, the user is presented with a choice of matching constraints (i.e. if they choose to create a constraint based on decimal or integer, a user can only choose to create a RangeConstraint). In the last step, the user has to enter the constraint boundaries, e.g. upper and lower bound or allowed values. It is always ensured that no one can insert invalid values (e.g. text into the upper bound) or override an existing constraint. As soon as a constraint has been established, a value type can be created by naming and choosing its built-in value type and its constraints. It is guaranteed that the user cannot match mismatching built-in value type and constraints.

Besides editing the data, a user can view informative statistics about the data in a column to get a better understanding of the content of the database. Those statistics depend on the value type, respectively built-in value type and they

change automatically when the value type is changed. A demonstration is shown in figure 5.3. The following statistics are always available:

- Number of entries
- Number of unique entries
- Frequency distribution

If the value type of a column is integer or decimal, more statistics can be computed:

- Sum
- Mean
- Median
- Data range
- Variance
- Standard deviation

These statistics are just a small selection of all possible statistics that could be integrated into this software.

Statistics for column "postleitzahl"

Number of entries: 17
Number of unique entries: 1
Sum: 307320
Mean: 18077.647058823528
Median: 18055
Data ranges from 18055 to 18182 (Range: 127)
Variance: 1246.8166089965398
Standard deviation: 35.310290412237336

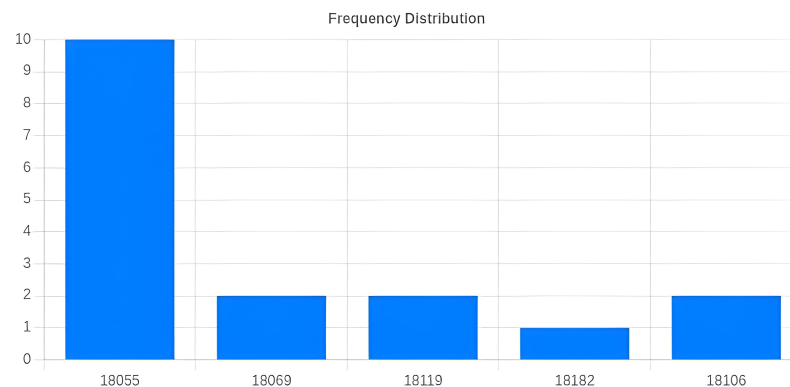


Figure 5.3: Example of statistics available for a column with integer value type.

In addition to modifying the data, it's also possible to filter the values using the search bar. This results in every row being searched for the requested value. The row is then temporarily hidden if the value is not contained by any of its cells. By clicking on the folder button, the user can view the project folder using the file explorer of the Operating System (OS). A project can easily be deleted by removing the folder from the workspace folder using the OS.

5.1.2 Backend

The backend processes all user requests forwarded by the UI. There are multiple checks and automations to ensure that a valid Jayvee script is generated:

- Check if folder exists: During the process of creating a folder inside the workspace, the software checks if the folder already exists to ensure that other projects will not be overwritten.
- Check URL: Examines if the URL points to a valid CSV file.
- Determine the encoding, delimiter and enclosing.
- Some CSVs have comments or empty lines at the beginning. The software tries to find and remove them.
- Extract column header: After detecting metadata like encoding, delimiter and enclosing, the Jayvee Data Wrangler extracts the column names. If the header could not be detected, it automatically assigns custom header names (column1, column2 and so on).
- Determine column value types: The software could use some heuristics, but to make sure nothing is missed, the software scans all entries of each column to detect the columns value types. This is also necessary because Jayvee does not support empty cells inside of columns with a value type other than text. Using a heuristic and not scanning the whole file could cause some rows to be skipped.

If at any point a check fails, the backend notifies the frontend component, which tells the user to change parameters or restart the modification. After validation, the changes are written into a Jayvee script, which is executed every time a user starts an import process or saves changes made to the database via the UI. While saving, the metadata database is also updated. If an error occurs, detailed instructions are sent to the frontend to inform the user about further steps to fix the problem. The user can also abort the changes by navigating back to the main page. There they can also open created projects to view and edit their database and Jayvee script. Of course, if the user opens a previously created project, the same editing options mentioned above are also available.

5.2 Interaction Between Components

This section visualizes how different components of the Jayvee Data Wrangler work together, while also describing how the user interacts with the software. Having in mind that the software should be easy to interact with and also provide features for experienced users, a typical interaction with the Jayvee Data Wrangler is shown in figure 5.4. The internal process of importing a CSV file into a database is illustrated in the sequence diagram. There are three main components, each of which has a different functionality. In the implementation, the parts are not so strictly separated and also consist of more components, but the diagram is kept simple for easy understanding.

The user interacts with the software's UI, while the UI communicates with the backend software that creates and executes the Jayvee script to import the data into a database. During this process the user is required to provide necessary data, such as the folder name that will also be used to name the database, the URL pointing to the CSV or any missing metadata. Every time the UI sends data to the backend, it is validated. If the data is invalid, the user needs to enter corrected data for the process to work. When the user starts the import process, the Jayvee script is populated with the previous user input and executed via the Jayvee interpreter. After importing the data, the user has the option of navigating to the folder to view and edit the Jayvee file, or simply viewing and editing the data in the software, as shown in the figure 5.5. It should be noted that in all the figures in this chapter minor details have been omitted to avoid confusing the reader. The demonstration illustrates the user action of requesting the user interface to display the data, which triggers the backend to retrieve the data from the database. The user can then perform a number of tasks:

- Edit the column name
- Change the value type
- Create a constraint
- Create a custom value type
- Display all created constraints and value types
- Delete rows
- Delete columns
- Navigate through the data
- Show the containing folder in the file explorer
- Undo changes

- Redo actions
- Save the modifications into the database
- Navigate back to the home page
- Navigate to the help page

When a user loads a previously created project, the interaction works in the same way as just shown, except that a list of projects is generated before from which the user chooses one to view and edit. It is noticeable that each time the backend is called upon to save, insert or update values, it executes an internal function to prepare the data for insertion into the Jayvee script when the user clicks the save changes button. To give an example, figure 5.6 shows the process of creating a new value type assuming that a matching constraint was previously created.

If a user requests to create the value type, the UI displays some input fields and prompts the user to enter the necessary data, including the option to add more than one constraint (this process has already been described in section 5.1.1). After filling all required fields, the data is sent to the backend, which stores the created value type internally. All the other non-trivial interactions are described in detail in section B within the appendix of this thesis. Some actions can be quite complex. For example, undoing an action (figure B.7 within the appendix) requires that the UI keeps track of all actions done and to execute different procedures depending on the last action.

If a row was deleted, the UI must undo all the visual modifications as well as to initiate the backend to update its record of deleted rows. The same applies if the last action was deleting a row. If a value type has been changed, there are two cases to consider. In the first case, the new value type allows more values; in the second case, the value type limits the number of valid values. The last action that can be undone is changing the name of a column, which loads the old column name which was saved when the user applied the new name. As always, the changes are also saved in the backend.

5. Architecture and Design

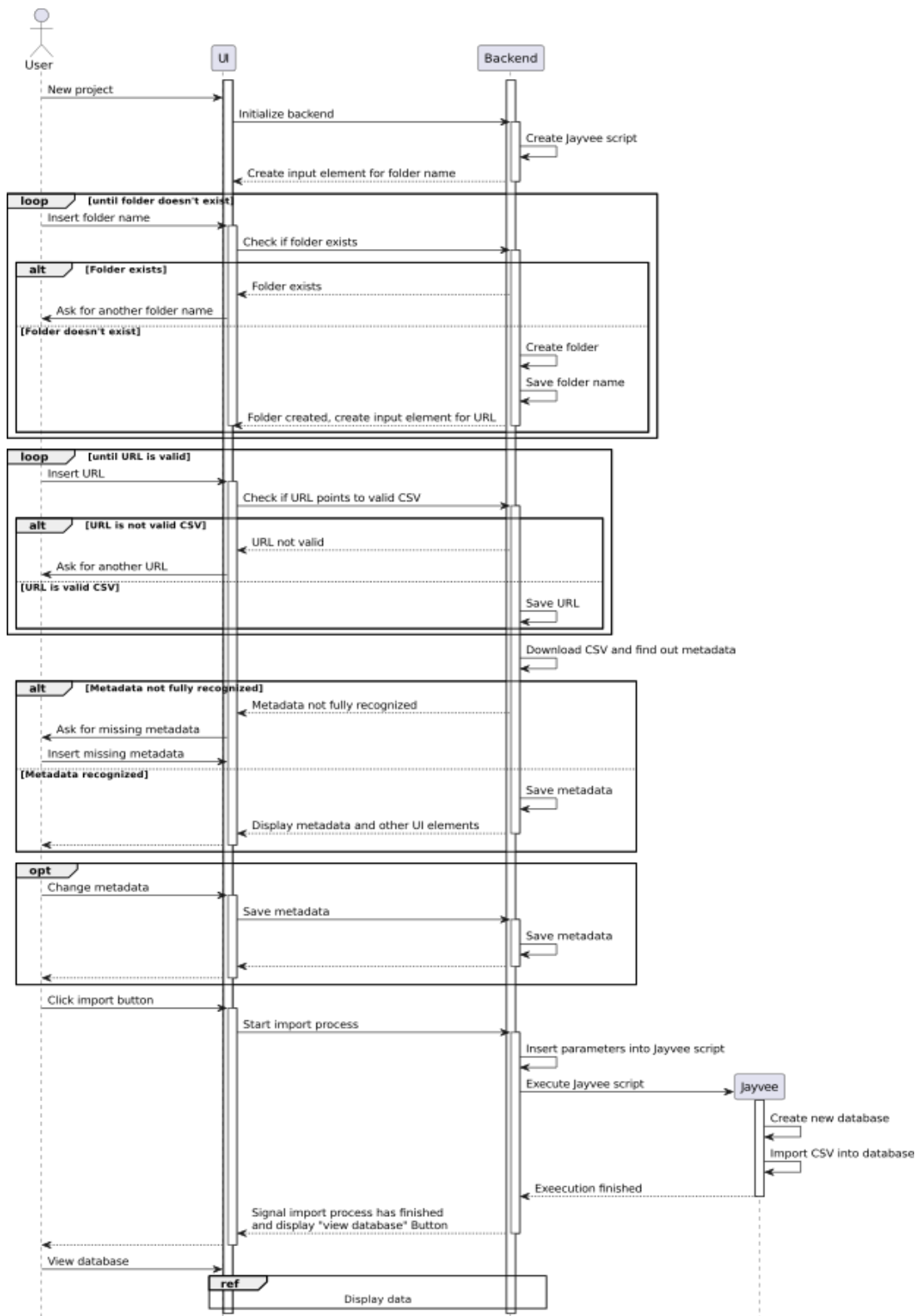


Figure 5.4: Interaction between the user and software, and between different parts of the software.

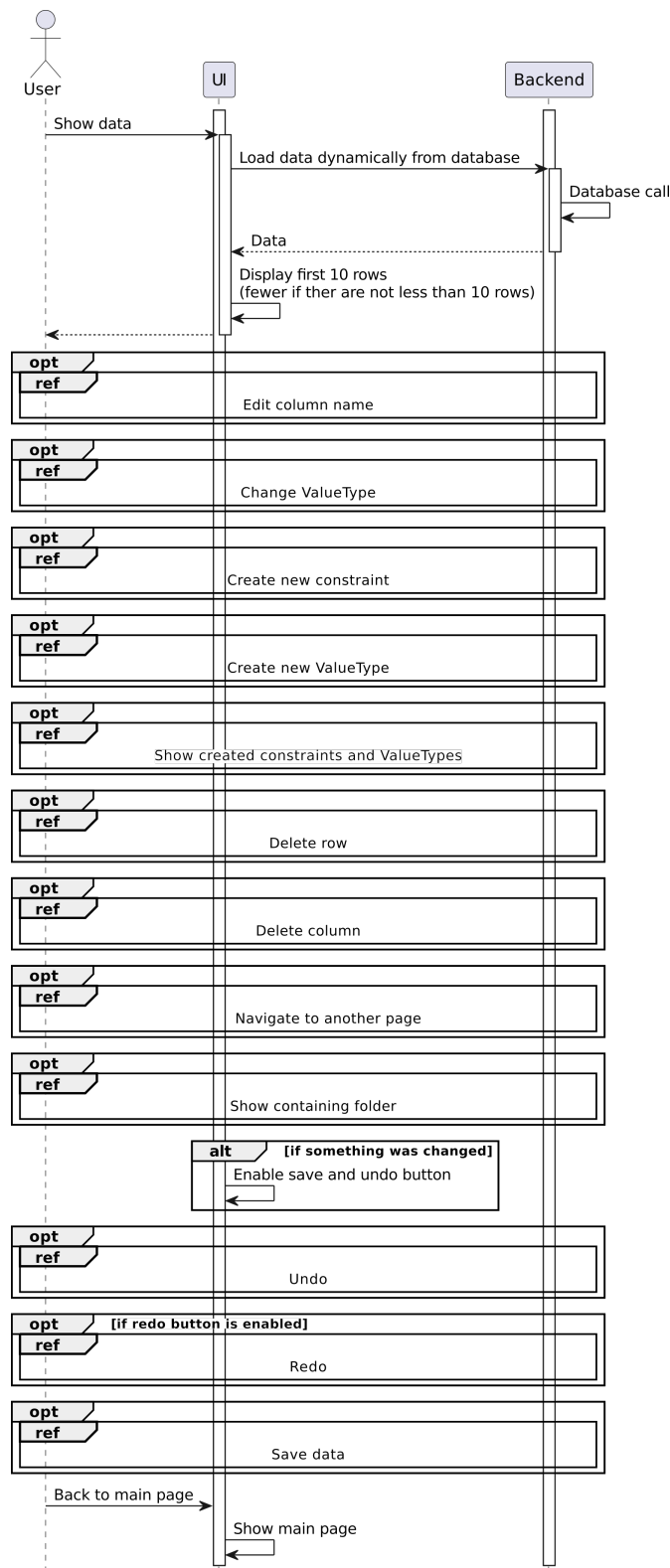


Figure 5.5: Interaction between the user and software, and between different parts of the software when the user displays the database.

5. Architecture and Design

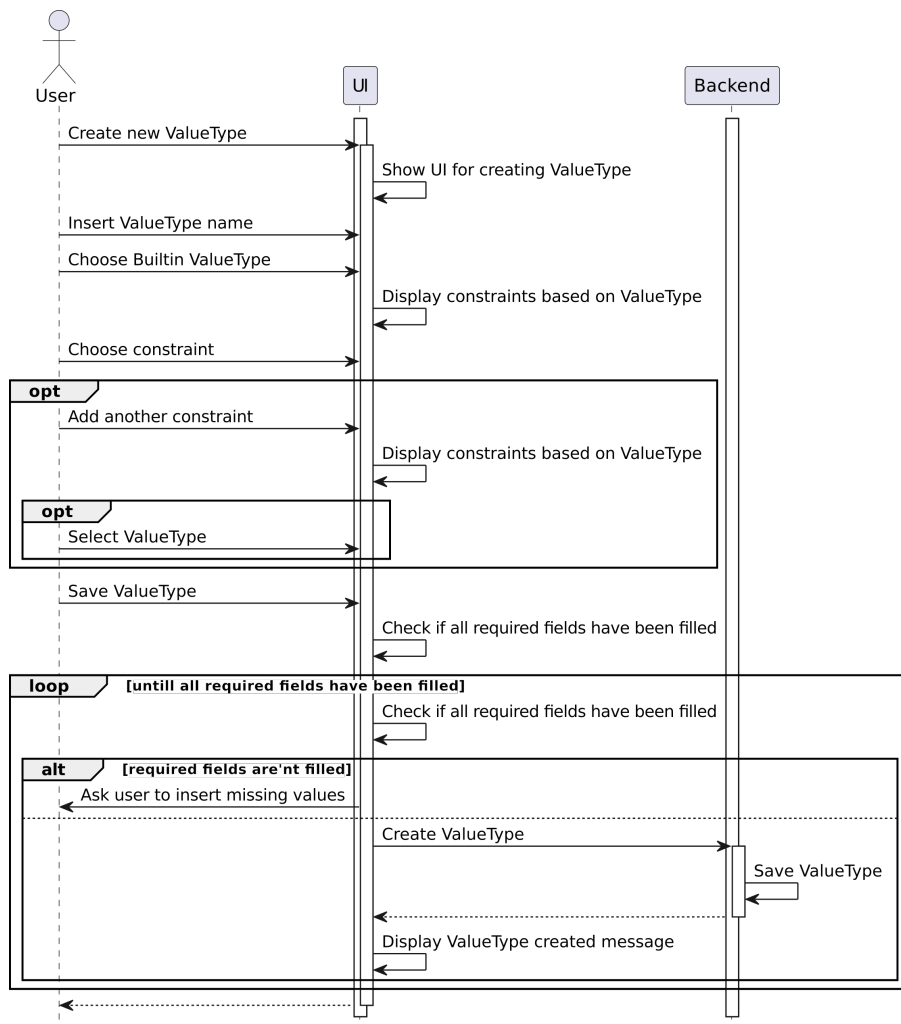


Figure 5.6: Interaction between user and software when creating a new value type.

6 Details of the Implementation

The Jayvee Data Wrangler is a cross-platform software developed in *TypeScript* using the framework *Electron.js* (hereafter referred to as Electron¹). The software consists of two main parts: *frontend* and *backend*. Both run within the Electron.js framework, while during the early stages of development a prototype of the background process — which works independently of Electron using command line input — was programmed. This prototype can be found in the thesis' GitHub repository. The frontend of the Jayvee Data Wrangler consists of several files controlling the UI and handle user input. All frontend files communicate with the background process which contains the algorithms that, among other things, create folders, extract metadata and save the Jayvee file.

In order to explain all the important components in detail, this chapter is divided into three parts. First, in section (6.1) the components and structure of the project, including the essential components that are within an Electron application, are introduced. Section 6.2 focuses on the import of CSV files into a SQLite database — including the detection of metadata and execution of Jayvee files —, while in section (6.3) the algorithms that enable and execute the modifications on the data are explicated. In the last section (6.4) some considerations about the license of the Jayvee Data Wrangler are made, as well as an outlook is given on how to create the executable application of the Jayvee Data Wrangler.

6.1 Components and Structure of the Project

The Jayvee Data Wrangler consists — besides from the framework — mainly of the files shown in figure 6.1. They are classified into frontend and backend to categorize the files which primarily control the UI, and those that have got to do with the main logic of creating, updating, executing and storing Jayvee files and databases. All frontend files consist of an HTML file and one or more JavaScript files which are not directly programmed, but compiled from TypeScript files. All backend files are developed in typescript and compiled to JavaScript. Figure 6.1 gives a better understanding of which files belong to each component and how

¹<https://www.Electronjs.org/>

6. Details of the Implementation

they depend on and interact with each other.

The components do not communicate with each other directly. Instead, they utilize an inter-process communication mechanism by using the *index.ts* file, which is intentionally omitted from the diagram to avoid unnecessary complexity. Both the backend and the frontend use their own specific helper functions to establish communication with each other and with the index file.

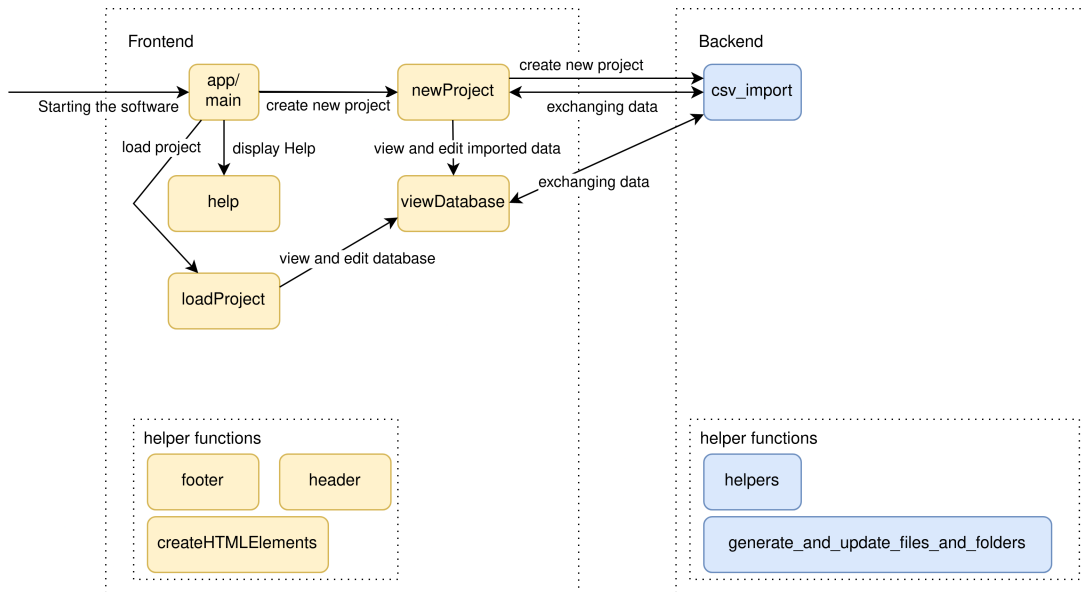


Figure 6.1: Communication between components of the frontend and backend.

The frontend consists of five main components and three helper files. The helper files contain the code that controls the UI. This will be described in detail in section 6.2. The backend also has helper files which are to create and modify files and folders (e.g. insert text into the Jayvee script or create the metadata database), and to analyze the CSV file. In addition to the files in the diagram, there is also an *app.css* file which defines the overall styling of the entire software and an *index.ts* file that handles the initialization of the Electron application, as well as some communication between the files. A preload script selectively enables features from libraries. Both features are explained later on. To avoid making the diagram more complicated, these three files are not illustrated.

As mentioned above, the Jayvee Data Wrangler uses the open source Electron framework to run as an independent software. According to their website (OpenJS Foundation, 2023a), Electron allows programmers to build cross-platform desktop applications using web technologies such as JavaScript, HTML and CSS. Each Electron application can be packed into different installers to run on Windows, Linux or macOS. Shipped with its own instance of Chromium and

having the cross-platform JavaScript runtime environment Node.js² (hereafter referred to as Node) included, it allows users to maintain a JavaScript codebase without requiring native development experience (OpenJS Foundation, 2023b). Due to the fact that Jayvee is also available via Node, it does not need to be integrated manually and Jayvee Scrips are easy to run. In an Electron project, users can integrate their favorite libraries and frameworks from the front-end ecosystem like Angular or React, or just develop their bespoke HTML code. Electron is widely used (many popular apps like Discord or Signal use it) since it is maintained and frequently updated (OpenJS Foundation, 2023a). Because of all these advantages and features, the Jayvee Data Wrangler is developed using Electron. No additional framework is used in order to maintain simplicity and clearness. Therefore, the software is written in TypeScript, HTML and CSS. Thanks to a config file, all TypeScript code is automatically compiled to JavaScript during development every time the application is launched or every time an installer is generated. Using TypeScript instead of programming directly in JavaScript makes the code not only easier to develop, but also easier to understand.

Electron behaves similarly to modern web browsers because it inherits its multi-process architecture from Chromium. More about that can be read in the documentation³. Briefly summarized, it has three processes: *Main*, *render* and *preload*. Details about electron explained below are taken from the official documentation (OpenJS Foundation, 2023b).

A single main process is responsible for interacting with the user. By generating web pages, it has the capability to present a graphical user interface. It also executes system-level operations and generates renderer processes.

Renderer processes display web pages within the application. Each page runs in its own render process. Typically, web pages in internet browsers operate within a restricted environment known as a sandbox. This setup restricts their access to native resources. Developers of Electron applications have the possibility to utilize Node.js APIs within their web pages, enabling them to perform lower level interactions with the operating system.

For security reasons, context isolation is used. This means that Node.js and Electron APIs are only available if they are preloaded by the preload script, which is run before the render process. The Jayvee Data Wrangler also has a fourth category of processes: the background processes. These processes execute Node.js scripts inside child processes in the background, completely separated from the main and renderer processes. For example. *csv_imports* and its helper functions run in a background process to interact with the file system.

Every Electron application has a main file which is loaded at the start. It is specified along with the packages used in a *package.json* file, which in case of the Jayvee Data Wrangler can be found in the GitHub project of this thesis. This

²<https://nodejs.org/en>

³<https://www.electronjs.org/docs/latest/>

file is active as long as the application is running. The `index.ts` file acts as the main process and creates render processes to load HTML files by default (when starting the application, see listing 6.2) or after receiving Inter-Process Communication (IPC) messages. In the provided code snippet (listing 6.1), an example of IPC is presented. Using IPC is an efficient and secure way to call APIs, such as the Electron or Node.js API, from the UI. The Jayvee Data Wrangler uses IPC not only to load files, but also to exchange data between the render process and files running in the backend.

The inter process communication works uni- and bi-directional. Following features of Electrons IPC are used within the Jayvee Data Wrangler:

- `send`: Send messages
- `on`: Listen for messages
- `invoke`: Send a message and wait for an answer
- `handle`: Receive a message and send an answer

While the main process has direct access to Electron and Node, the render processes require those functions to be called via `ipcMain` or being preloaded via a preload script (the script used in the Jayvee Data Wrangler is in the appendix section C). This script exposes some functions to the main window, which can then be accessed by a variable that all render processes can use to call the functions. Further details about IPC and preload scripts can be found in the documentation⁴.

```
1 // Routing to open new pages.
2 ipcMain.on("newProject", () => {
3   mainWindow?.loadURL(
4     `file://${path.join(__dirname, "src", "newProject", "newProject.html")}`
5   );
6 });
```

Listing 6.1: Within the software Inter-process communication (IPC) is used to receive messages to e.g. swap the html file of the main process.

⁴<https://www.electronjs.org/docs/latest/tutorial/ipc>

```

1  let mainWindow: BrowserWindow | null;
2  let db: sqlite3.Database;
3  function createWindow() {
4    // Get the primary display size.
5    const { width, height } = screen.getPrimaryDisplay().workAreaSize;
6
7    // Create the browser window.
8    mainWindow = new BrowserWindow({
9      width: width,
10     height: height,
11     autoHideMenuBar: true,
12     webPreferences: {
13       preload: path.join(__dirname, "preload.js"),
14       nodeIntegration: true,
15     },
16   });
17
18   // Load the startpage of the app.
19   mainWindow.loadFile(path.join(__dirname, "src", "app.html"));
20
21   // Emitted when the window is closed.
22   mainWindow.on("closed", function () {
23     // Dereference the window object.
24     mainWindow = null;
25   });
26 }
27
28 // This method will be called when Electron has finished
29 // initialization and is ready to create browser windows.
30 app.on("ready", createWindow);

```

Listing 6.2: Creating the window of the application and loading the startpage.

6.2 Importing CSV files

6.2.1 Generating and Modifying the UI

The UI always has the same structure (figure 6.2). Each page has the same header and footer. The header contains the Jayvee Data Wranglers logo and a main menu that allows the user to navigate to the help section and also back to the main page. The IPC is used to navigate to other pages. The *header.ts* sends a message to the IPC channel. Inside *index.ts* a listener waits for the message and loads the home file into the main process. The footer contains information about the author and the license. In between these components the UI is dynamically generated using functions from *createHTMLElements.ts*.

Within *createHTMLElements.ts* there are functions for creating *HTMLElements* like paragraphs, editable paragraphs, drop-down elements, buttons, dialogs or bar charts for statistics. The styling of the UI-elements is within a single file (*app.css*). To give an example, one of the functions will be explained below. All functions can be found well documented on GitHub.

The *createEditableParagraph* function creates a paragraph element with buttons that becomes editable on click by displaying an input element. The user can

6. Details of the Implementation

modify or replace the current value and save it, cancel changes or restore the initial value. The steps which the function has to execute when called are annotated in listing 6.3, while the code has been expanded for sake of clarity.

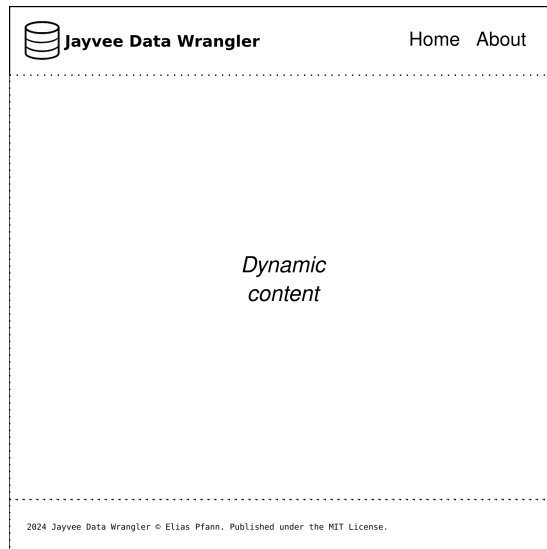


Figure 6.2: The consistent construction of the Jayvee Data Wrangler.

```
1 /**
2  * @param {string} id – The id of the paragraph element or nulls.
3  * @param {string} text – The initial text content.
4  * @param {RegExp} allowedChars – The allowed characters for the input.
5  * @param {number} maxChars – The number of characters allowed in the input.
6  * @param {function} saveChanges – Function to call when save is clicked.
7  * @returns {HTMLDivElement} – The created paragraph element container.
8  */
9 export function createEditableParagraph(id: string, text: string,
10   allowedChars: RegExp, maxChars: number, saveChanges: () => void):
11   HTMLDivElement {
12   // Create container element which holds the paragraph, input field and
13   // buttons.
14   // Create paragraph element, input element, edit, restore, save and
15   // cancel button.
16   // Function that shows the input when edit is clicked.
17   // Function to save the input if it does not contain restricted
18   // characters and to display a warning otherwise.
19   // Function to restore the original value.
20   // Function to abort the changes and restore the original value.
21   // Functions to handle enter and escape key press.
22   // Functions to show and hide UI elements and to add listeners for enter
23   // and escape key.
24   // Return the container.
25 }
```

Listing 6.3: The function `createEditableParagraph` creates a paragraph element which can be edited. To avoid confusion, only the descriptions are shown and some similar or trivial code sections are omitted.

These functions are mainly used within the files *newProject.ts*, *viewDatabase.ts* to dynamically create and update the UI. At the start of a newProject the process of importing a CSV is automatically initiated by calling the function *createNewProject* within the file *csv_import.ts* using IPC after loading *newProject.html* (listing 6.4).

```

1 newProject.ts:
2 // Ensure DOM content is loaded before accessing elements.
3 document.addEventListener('DOMContentLoaded', (event) => {
4   // Start the creation of a new project.
5   window.electron.send('createNewProject', null);
6 });
7
8 index.ts:
9 import { createNewProject, cleanup } from './src/dataWrangler/csv_import';
10 // Start the process of creating a new project.
11 ipcMain.on('createNewProject', async (event) => {
12   try {
13     // The UI has to be cleaned up,
14     // e.g. remove HTML Elements from previous imports or reset variables.
15     await cleanup();
16     await createNewProject();
17   } catch (error) {
18     // Send error back to the renderer process.
19     throw new Error((error as Error).message);
20   }
21 });

```

Listing 6.4: Starting the process of importing a CSV into a database using IPC.

The UI is now controlled by the code inside the background process *csv_imports.ts*. Every time a user saves or restores data, clicks a button to start the import process or to view the database, *newProject.ts* sends the values or instructions to the background process which in return, sends messages to create HTML elements, which then are created within *newProject.ts* using *createHTMLElements.ts* and attaching them to the *newProject.html*. The diagram in the previous chapter (figure 5.4) allows retracing the aforementioned information. In this diagram, the background process is represented as the backend, while the rendering process is represented as the UI.

6.2.2 Analyzing CSV files

Creating a new project involves initializing the project folder, detecting metadata like encoding, delimiter and enclosing, as well as creating and running the Jayvee file. Recalling the diagram from figure 5.4 into memory a working folder has to be created before analyzing the CSV file. All project folders are created within a folder called *workspace* which is located inside an application folder (*jayvee-data-wrangler*) in the users app-data directory. This would typically be, for example, *~/config/jayvee - data - wrangler/* on Linux.

6. Details of the Implementation

To receive various inputs like the folder name, an asynchronous function (listing 6.5) was developed which listens once on the IPC channel `sendDataToCSVImports` and returning a promise which resolves with the message (user input). Asynchronous functions are used in Typescript/JavaScript to wait for a return value of a function before continuing (Mozilla Foundation, 2024).

```
1  /**
2  * Function to await data which is needed to continue the workflow.
3  * @returns {Promise<string>} A promise that resolves with the element to
4  * which the data belongs, along with the data.
5  */
6  async function waitForData() {
7      return new Promise((resolve) => {
8          // Define the IPC event handler.
9          const handler = (event: any, [input, data]: [string, any]) => {
10             // Process data or perform actions here.
11             console.log('Received data:', data);
12             // Remove the listener to prevent memory leaks.
13             ipcMain.off('sendDataToCSVImports', handler);
14             // Resolve the promise with the received data.
15             pipeline[input] = data;
16             resolve([input, data]);
17         };
18         ipcMain.on('sendDataToCSVImports', handler);
19     });
20 }
```

Listing 6.5: Function to listen on an IPC channel to receive data from other files.

All received data will be stored within a pipeline object (see appendix C, section C.2). Because there is a certain sequence of events, it is always known where the data belongs to and therefore, exactly where it must be stored. The data from this object will be stored in the Jayvee script if the user clicks the `import csv` button or saves changes made to the database.

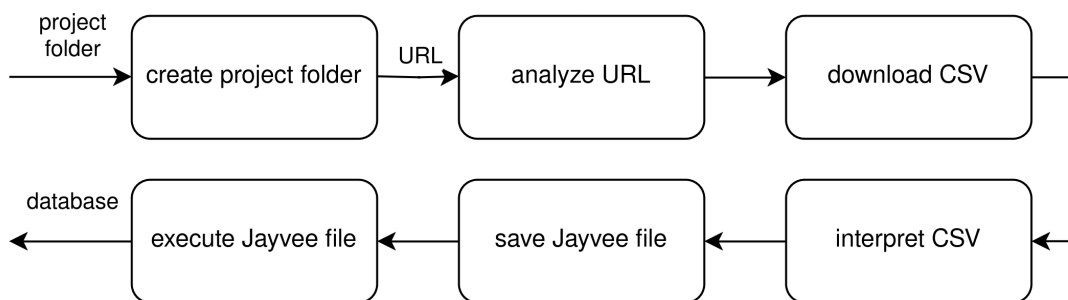


Figure 6.3: Steps during the process of importing a CSV into a database.

The code for creating a directory, modifying files or creating databases can be

found inside *generate_and_update_files_and_folders.ts*. Algorithms that analyze the CSV file or modify parameters before saving them in the Jayvee file are inside *helpers.ts*. Before the contents of the CSV can be viewed, a project folder must be created, the URL has to be checked for validity and the CSV file must be downloaded and analyzed to determine the metadata.

To create a directory, the Node modules *path* and *fs* are used to determine the user's path and to create the project folder (listing C.3 of appendix C). At each step, if the algorithms detect something unsuitable, e.g. if the folder name already exists, an IPC message is sent to *newProject.ts*. Thereafter, the user has to correct their input.

To analyze the CSV file, it is downloaded and thereby checked for validity. Among other things, the Node library *Axios* is used to make HTTP-requests. Each URL is validated by checking the file header. Because CSV files may be saved in a format other than CSV, the file is also checked for other schemas, as well as the file extension. This pre-validation prevents that URLs are pointing to non-CSV files (e.g. PDF). It is important to note that there is a timeout of one minute in order to prevent any issues.

After the file is downloaded, it is interpreted. This includes the following in the order listed:

1. Identify the encoding
2. Detect comments
3. Extract preview data
4. Identify the delimiter
5. Identify the enclosing
6. Extract the header
7. Rename duplicates within the header
8. Create header-letter mapping
9. Determine value types of the columns

The encoding is identified using the Node library *chardet* and checked to see if it is supported by Jayvee. If it is not detected or supported, the user has to choose a different encoding.

Comments could most likely occur at the beginning of a CSV file. They are detected scanning the file line by line and comparing if the line starts with one of the common comment indicators ('#', '//', ';', '-', '/*', '!', '%', '[') until a line has no comment indicators. Empty lines between comment lines are also recognized. Not only is the number of comment lines saved for the Jayvee pipeline, but the comment lines are also stored in a text file in the project folder so that the user can view them.

6. Details of the Implementation

To identify encoding, delimiter and enclosing manually, a preview of the data is generated. For that, the first row of the CSV is read by skipping the comment lines, empty lines and header. If the CSV has no header, the second line is read. Identifying the delimiter takes advantage of the fact that a character occurs frequently in a row if it is the delimiter, but not very often if it is part of an entry. By reading the first line of the file (after skipping comments and empty lines) and checking which potential delimiter (',' , ';' , '\t' , '|' , ':' , ' ') occurs the most (listing 6.6), the delimiter can usually be identified.

```
1  const potentialDelimiter = [ ',', ';', '\\t', '|', ':', ' ' ];
2  let bestDelimiter: string | null = null;
3  let maxDelimiterCount = 0;
4
5  // Count the occurrences of each delimiter in the sample lines.
6  for (const delimiter of potentialDelimiter) {
7      if (sampleLine) {
8          const delimiterCount = sampleLine.split(delimiter).length - 1;
9          // Update the best delimiter if the count is higher.
10         if (delimiterCount > maxDelimiterCount) {
11             maxDelimiterCount = delimiterCount;
12             bestDelimiter = delimiter;
13         }
14     }
15 }
```

Listing 6.6: Excerpt from the function *identifyCSVDelimiter*. It shows the procedure of finding out the delimiter by checking the frequency of potential delimiters.

The enclosing character is determined using the same frequency check procedure with two potential enclosing characters: " and '. Later, knowing the enclosing, it will be stored in the Jayvee file using the respective other character to identify it as a character (e.g. " will be stored as "'').

It is often challenging to definitively determine whether the first row of a CSV file serves as the header or as the first row of data. Nevertheless, there are scenarios where the initial line can be disregarded. Given that the header descriptions usually do not contain numeric values and numeric header values are also not supported within Jayvee The JValue Project, 2024, a row containing a numeric value cannot be the header. The code within the respective function (*getHeader*) therefore, checks every entry of the first line (skipping comments and empty lines) if it is not a number. The entries are identified by separation through the delimiter. This is also the main reason why the order of the interpretation step is essential. If a header contains duplicates, they are renamed by adding an underscore and a current number. If no header is detected, one is generated by naming the columns like this: col1, col2, ...

Having identified the header, a header-letter mapping is created. This mapping from column headers to Excel-style column letters is used later on to identify

columns within the Jayvee file. For example, deleting a row in Jayvee requires inserting the letter of the deleted row (The JValue Project, 2024).

The last step before executing the Jayvee file is the most complicated. The value types of a column must be determined. Detecting the wrong value type, as well as making any other mistakes before, would most likely lead to the crash of the execution of the Jayvee pipeline. That is why in the worst case the function for identifying the value type has to scan the whole CSV file.

The function (*determineColumnValueTypes*) reads the file line by line, skipping comments and empty lines. If a column was already assigned with the value type text, then it logically cannot be assigned with any other value type, so those columns are skipped. If a value type is not determined yet, it is tested in various ways. The order of the tests is essential, because e.g. testing for a text before testing for an integer would cause all integer values (as they are also text values) to be text. The following conditions are listed in the order they are used to check a value type. If a value fulfills the first condition its value type is text, if not the second condition is checked (and so on):

- if value == " " → text, because no other value type is capable taking entries with empty cells.
- if value.toLowerCase() == "true" || value.toLowerCase() == "false" → boolean
- const numberRegex = /^[+-]?([0-9]*[.])?[0-9]+([eE][+-]?\d+)?\$/; if numberRegex.test(value) → integer
- const decimal = Number.parseFloat(value.replace(',', '.')); const integer = Math.trunc(decimal); if decimal == integer && columnValueTypes[index] != "decimal" → If the column not already contains decimal values it can be of type integer, else its value type is decimal.
- If it is not determined yet, the value type is text.

If all columns are assigned with the value type text, the algorithm stops, because once a column's value type is detected as text, no other value type can be assigned. As soon as the value types are detected, *newPage.ts* receives a message to create a button to run the Jayvee pipeline. When it is clicked, the previously saved data is written into the Jayvee script. During this process, clicking on the UI is disabled. Because the function shown in listing 6.7 is also used when modifying the database, it can insert constraints and value types, and delete rows and columns. The function also stores the metadata which cannot be restored from the Jayvee script or database in a separate SQLite database and executes the script to import the CSV into a SQLite database using the Node module *child_process* that allows the scripts to be executed.

6. Details of the Implementation

```
1 ipcMain.on('pipelineStart', async () => {
2   // Create the Jayvee file.
3   await fileHelpers.writeFile(pipeline.directory, "pipeline.jv",
4     'pipeline TestPipeline {Extractor -> TextFileInterpreter->RangeSelector
5     ->CSVInterpreter->ColumnDeleter->RowDeleter->TableInterpreter->Loader;
6     block Extractor oftype HttpExtractor{ url: "${pipeline.url}";}'');
7   // Create the pipeline.
8   await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
9     'block TextFileInterpreter oftype TextFileInterpreter {
10    encoding: "${pipeline.encoding}";}'');
11  // Remove comments at the beginning of a file.
12  await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
13    'block RangeSelector oftype TextRangeSelector {
14    lineFrom: ${pipeline.commentLines + 1};}'');
15  // Enclosing and delimiter.
16  [...]
17  // Remove cols and rows.
18  [...]
19  // Create the table interpreter.
20  await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
21    'block TableInterpreter oftype TableInterpreter {
22    header: false;
23    columns: ['');
24  for (let i = 0; i < pipeline.header.length; i++) {
25    if (i == pipeline.header.length - 1) {
26      await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
27        "${pipeline.header[i]}" oftype ${pipeline.value types[i]};}'');
28    } else {
29      await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
30        "${pipeline.header[i]}" oftype ${pipeline.value types[i]};}'');
31    }
32    // Store metadata in the database for e.g. loading the project.
33    [...]
34  // Store the database connection.
35  await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
36    'block Loader oftype SQLiteLoader {table: "${pipeline.table}";
37    file: "${pipeline.databasePath}/${pipeline.database}.sqlite};}'');
38  // Close the pipeline.
39  await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv", ' ');
40  // Create value types.
41  for (let value type of pipeline.createdvalue types) {
42    await fileHelpers.appendToFile(pipeline.directory, "pipeline.jv",
43      'value type ${value type[0]} oftype ${value type[1]}
44      {constraints: [${value type[2].join(', ')}];}'');
45    // Store metadata in the database for e.g. loading the project.
46    [...]
47  }
48  [...]
49  // Create constraints.
50  [...]
51  // Execute the pipeline.
52  const filePath = (path.join(pipeline.directory, "pipeline.jv"));
53  exec('jv ${filePath}', (error) => {
54    if (error) {
55      console.log('error: ${error.message}');
56      mainWindow?.webContents.send('pipelineFinished', true,
57        [null, null, null]);
58    } else {
59      mainWindow?.webContents.send('pipelineFinished', false,
60        [pipeline.directory, pipeline.database, pipeline.table]);
61    }
62  });
63 });
```

Listing 6.7: Function to create the Jayvee Script. To keep it clear and concise, only some parts are shown serving as an example.

If an error occurs during the import, the user is prompted to modify the metadata. To find out what could have gone wrong, they can open the project directory in the default file explorer and have a look at the CSV or the Jayvee pipeline. Opening the folder is delegated to the main process using IPC. The folder is opened using Electrons *shell* module.

After a successful import, the user can view the data by clicking on the newly generated button. The database view is generated within *viewDatabase*. To know the location of the database and know which table to display, these parameters are passed encoded in the URL (appendix C.4). This is possible because Electron uses a browser environment.

6.3 Modifying data

After the user finishes the import of the database and proceeds to view the database, *viewData.html* is displayed. Within its TypeScript file, UI elements are also loaded via the helper functions from *createHTMLElements.ts* except from creating the view of the database. This view is created using the jQuery JavaScript library *datatables*⁵. Supporting pagination, search and other features, this highly flexible tool facilitates the display of the data.

To use *datatables*, the data has to be loaded from the database first. This is done using the *SQLite3* function *each()*. This function executes an SQL statement and returns the results line by line (Lockyer, 2022). The database is loaded in chunks and only 100 lines are loaded. This increases performance and enables the user not having to wait until everything is loaded and enables the user to interact with the datatable. This is very efficient on large databases. If the user clicks on the next button within the datatable, another 100 rows are loaded (see listing 6.8. Loading all rows at once would block the user interface and could cause unexpected wait times.

All database calls are, invoked via IPC, made within the main process. Bootstrap⁶ is used to determine when the page has finished loading. Immediately after that, the UI is generated including populating the datatable (appendix C.5 and C.6) and creating a toolbar with buttons for saving the content, performing undo and redo, creating new constraints and value types, as well as displaying them.

To get the header of the table, another database call is invoked. The functions within *viewDatabase.ts* are also used to display the data when a previous project is loaded. Therefore, also metadata in the background process (*csv_import.ts*) has to be restored, as it might not be present. Value types are then loaded from *csv_import.ts* using IPC communication and added into its own row below the header names. Below that buttons for deleting columns, viewing statistics and

⁵<https://datatables.net/>

⁶<https://www.npmjs.com/package/bootstrap>

6. Details of the Implementation

switching the value type are added, as well as a column is added to contain buttons for deleting rows.

```
1 // Load the data from the database. The data is loaded in chunks of 100 rows
2 // to improve performance.
3 let pageSize = 10;
4 let offset = 0;
5 setTimeout(async () => {
6   await dbEach('SELECT rowid,
7     * FROM "${tableName}" LIMIT ${pageSize*10} OFFSET ${offset}');
8   offset += pageSize*10;
9 }, 1);
10
11 // If the user clicks on the next button, nerxt 100 rows are loaded.
12 dataTable.on('page', async function () {
13   await dbEach('SELECT rowid,
14     * FROM "${tableName}" LIMIT ${pageSize*10} OFFSET ${offset}');
15   offset+=pageSize*10;
16 });
```

Listing 6.8: Dynamical loading of data into the datatable.

Creating a constraint or value type is straightforward and involves creating UI elements and validating the data using regular expressions. When the user finishes the creation process, the constraint or value type is saved in an array and — as every created or modified value — send to the background process to be stored into the Jayvee script on saving the database. A new value type can immediately be used within the datatable. Editing the header name and displaying statistics (they are computed based on the value type or Built-in Value Type if the value type is a custom generated one) is as well straightforward.

As it can be seen in the appendix C.5 (lines 45 and 46), every time the value type is changed, two functions are called: *checkValueType* and *modifyDisplay*. They are also called during undo and redo actions, and if the user navigates to another page. Both functions iterate through the displayed values row by row and are called again if the user navigates to another page of the datatable which makes them quite performant. *modifyDisplay* is also called when a row is deleted. Both functions are used to modify the display of the data. The data is directly modified within the data table to recreate the process that Jayvee performs when the modified script is executed and to visually indicate changes by crossing out values.

The software does not keep track of the deleted values, but of the deleted rows, as deleting a value in Jayvee leads to deleting the entire row. It also keeps track of which columns lead to the deletion of a row, so that all values are checked again if that specific rows value type is changed. To illustrate that, an example is given in figure 6.1. The function *checkValueType* examines if the value matches the value type of the column (or if the built-in value type fits). If it does not match, its row is marked as crossed out by saving it to a map containing all crossed out

rows. This map is also used to store crossed-out rows if the user deletes a row. If the user switched to a custom value type and its built-in value type fits, the constraints are checked. In listing 6.9, there is a demonstration of how a value is validated against a constraint. In this example, after changing the value type of two columns, the *crossedOutRows* map would contain $\{3 \Rightarrow ['2', '3']\}$. Changing e.g. column 3 back to decimal would therefore not lead to a visual change for the last row, as row 2 would still be in the map ($\{3 \Rightarrow ['2']\}$).

	Name	Age	Income	Raise
value type	text	integer	decimal	decimal
1.	Lisa	18	25000	5
2.	Robert	49	96000	2.9
3.	Tom	37	5135,50	3.8

	Name	Age	Income	Raise
value type	text	integer	integer	integer
1.	Lisa	18	25000	5
2.	Robert	49	96000	2.9
3.	Tom	37s	5135,50	3.8

Table 6.1: Example showing the change of the value type of two columns.

```

1  if (constraintType?.includes('Denylist')) {
2    const deniedValues = constraintData?.[3];
3    if (deniedValues) {
4      const deniedValuesArray = deniedValues.split(',');
5      if (deniedValuesArray?.includes(cell)) {
6        if (!crossedOutBy.includes(dropdown.id))
7          crossedOutBy.push(dropdown.id);
8      } else { // Cell value fits the type of the column.
9        crossedOutBy = crossedOutBy.filter((value)
10         => value !== dropdown.id);
11        crossedOutRows.set(rowid, crossedOutBy);
12      }
13    }
14 } else if (constraintType?.includes('Length')) {
15   const min = Number(constraintData?.[3]);
16   const max = Number(constraintData?.[4]);
17   if (cell.length < min || cell.length > max) {
18     if (!crossedOutBy.includes(dropdown.id))
19       crossedOutBy.push(dropdown.id);
20   } else { // Cell value fits the type of the column.
21     crossedOutBy = crossedOutBy.filter((value)
22      => value !== dropdown.id);
23     crossedOutRows.set(rowid, crossedOutBy);
24   }
25 }

```

Listing 6.9: Example of validating values against constraints.

6. Details of the Implementation

Modifying the style of the values follows the same process. The style of a column is reset if the function is called within the undo function and a deletion of a column is undone. In other cases, the algorithm checks if the value was only crossed out by this column and now is no longer crossed out. Then its style is reset. In all other cases, if the value does not match the value type, the function modifies its style. A particularity is that columns that are deleted and already crossed out are skipped. This can be observed in the example shown in listing 6.10.

If at any time modifications are made, the toolbar buttons are also enabled/disabled. For example, changing a value type enables the save and undo buttons, but disables the redo button as the state from which a redo could be performed has been overridden.

```
1 allRows.forEach((row) => {
2   // Get the cell, +1 because of the index column.
3   const cell = (row as HTMLTableRowElement).cells[index + 1];
4   // Reset the row style only if it is not crossedOut by another column.
5   const rowid = (row as HTMLTableRowElement).cells[0].innerText;
6   let crossedOutBy = crossedOutRows.get(Number(rowid));
7   if(crossedOutBy && crossedOutBy.length < 1){
8     // Dropdown is the only one crossing out this row -> reset style
9     for(let i = 1; i < (row as HTMLTableRowElement).cells.length; i++){
10      // Skip the current column,
11      // if it was deleted or if the row was deleted.
12      if(deletedColumns.includes(i) ||
13         deletedRows.includes(Number(rowid))) continue;
14      (row as HTMLTableRowElement).cells[i].style.textDecoration =
15      'unset';
16      (row as HTMLTableRowElement).cells[i].style.color =
17      '#000000';
18    }
19  }
20  // If the cell value does not match the selected type, cross it out.
21  if(basevalue.types.includes(dropdown.value) ||
22     dropdown.value === 'boolean'){
23    // Remove empty entries if the value type is not text.
24    if((dropdown.value !== 'text' && cell.innerText === '') ||
25       !matchesType(cell.innerText, dropdown.value)){
26      // Cell of column is empty.
27      for(let i = 1; i < (row as HTMLTableRowElement).cells.length; i++){
28        // Skip the current column, if it was deleted.
29        if(i === index + 1 && deletedColumns.includes(i)) continue;
30        (row as HTMLTableRowElement).cells[i].style.textDecoration =
31        'line-through';
32        (row as HTMLTableRowElement).cells[i].style.color =
33        '#cccccc';
34      }
35    }
36  }
```

Listing 6.10: Excerpt from the algorithm that modifies the display of values.

Another approach to modifying the database would be to apply changes directly to the Jayvee file and execute it. This would make the algorithms much shorter at first glance, but would result in longer run times and delays, because not only

the pipeline needs to be executed, but all data has to be reloaded every time a change has been made. Also, changes would be applied to all data, not just the data that is currently displayed on the screen, which is quite inefficient. If, as it is implemented in the Jayvee Data Wrangler, the changes made to the data should be visible and restoring to a certain point possible, the algorithms would also become quite complex.

When the user clicks undo, the button listener is invoked and restores the previous state. The algorithm distinguishes between the actions performed.

- Column removed: Undo the cross out of the column and remove the row from the deleted columns record.
- Row removed: Undo the cross-out of the row and remove the row from the deleted rows record.
- Header name changed: Restore the previous header name.
- Value type changed: Change the value of the corresponding dropdown, check the value type for this row and modify the display of values.

After that, the action which was undone is saved in an array, so that the user can redo it and the changes are also sent to `csv_imports.ts`. Redo works similar to undo, except that it must — of course — do the opposite of each action. The code for the procedure described can be found in the GitHub repository associated with this thesis.

Finally, when the user clicks on save, the backend is invoked via IPC and saves the modifications into the Jayvee file and executes it. The user will be notified within the UI if saving was successful or an error occurred. The user can then do further modifications or navigate back to the main menu, where they can create a new project or view existing projects. If an error occurs, they can change the metadata to avoid the error or cancel the changes by navigating back to the home page (and start again). Existing projects can be deleted by removing the folder via the file explorer. If a user edits the database using other tools, they must be aware that the metadata database may also need to be edited if metadata has been edited (e.g. value type changed) or columns have been deleted. Otherwise, the project may not load using the Jayvee Data Wrangler. The Jayvee script can be edited by other tools because it is not used to load values, but is overwritten when the Jayvee Data Wrangler is used to edit the database.

6.4 License Considerations

On one hand, the Jayvee Data Wrangler should be accessible to everyone and extensible, and its code is reusable for everyone. Nevertheless, no guaranties are made to users out of precaution matters. Therefore, its code is published as open

6. Details of the Implementation

source project under the MIT license ⁷, which grants the rights of modification and redistribution without any charge, but does not give any warranties. When using the executable application version of the Jayvee Data Wrangler, the software is easy to use without programming experience (as already described in the previous sections and chapters). If the Jayvee Data Wrangler were available for download as a ready-to-use application — including all the libraries it uses (and therefore the libraries used within them) — it would have to be checked for the correct license(s), which requires advanced legal expertise (GitHub, Inc., 2024). Thus, the instructions for building this application are provided in the README of the GitHub repository, but the application is not available as executable, because it would contain 3rd party code.

⁷<https://opensource.org/license/mit>

7 Evaluation

In this chapter, the Jayvee Data Wrangler's evaluation is presented. First, the fulfillment of the requirements is checked in section 7.1. Then, some tests demonstrate the functionalities of this software (section 7.2). In the final section 7.3 of this chapter, the software is classified and compared to other software already shown in chapter 2.

7.1 Requirements Evaluation

The Jayvee Data Wrangler is evaluated in terms of fulfilling the requirements stated in chapter 4. First, the functional requirements are evaluated. Table 7.1 indicates which prerequisites were met.

The functional requirements 1-5 are fulfilled. Necessary explanations are provided below.

FR-1: All sub-requirements are fulfilled up to the point to which the user inserts invalid data. The software contains algorithms that analyze the URL in order to ensure that a valid CSV will be imported into a database. Other algorithms recognize all the metadata required to create and execute the underlying Jayvee script. In case the metadata can not be recognized, the user has to insert it. Otherwise, the software will not continue. Of course, there is a minor chance that the software recognizes some metadata incorrectly, which could lead to the import of wrong data. Also, — in case the user inserts incorrect metadata — the software may not work as the underlying Jayvee script may not execute properly.

FR-2: The user can edit all the parameters stated in FR-2.1-2.4. As updates of the database (when changes are saved by the user) are done via an execution of a modified Jayvee script, all concepts of Jayvee are complied.

FR-3: Each CSV file is imported separately and a separate script is generated for each of them as soon as the user clicks on the import button. Updating the database presupposes the update of the Jayvee script, since the database is updated by executing an updated Jayvee script. All Jayvee scripts are editable as they are stored in separate files on the users device.

FR-4: The UI is kept minimalistic. The user will be guided step by step through the import of a CSV file. All actions a user can perform are also explained within the help section of the software.

FR-5: The user can generate and display statistics via one simple click of a button. These statistics also vary when changing a column's value type. Statistics for deleted columns are disabled.

FR-6 is fulfilled up to a certain point, because all input fields are either restricted to a set of characters that the operating system or Jayvee accepts, or are checked for invalid inputs before proceeding (with a warning being displayed). When the user enters wrong metadata, it could be considered as invalid input, which would violate this requirement. Unfortunately, it is out of the programmer's reach to avoid human (users) mistakes.

The non-functional requirements are mostly fulfilled: NFT-1 is by using only publicly available libraries from Electron and Node.

NFR-2 is also fulfilled as it does not depend on proprietary libraries and is published under the MIT license.

NFR-4 is partially fulfilled. While NFR-4.1 and 4.2 are fulfilled, NFR 4.3 is so only partially, as most of the important components are annotated with comments, but not all. There is also external documentation in the GitHub README.

ID	Fulfilled	ID	Fulfilled
FR-1	yes	NFR-1	yes
FR-2	yes	NFR-2	yes
FR-3	yes	NFR-3	yes
FR-4	yes	NFR-4	partially
FR-5	yes		
FR-6	yes		

Table 7.1: Summary of requirement fulfillment

7.2 Examples of Testing

To evaluate the functionality of the Jayvee Data Wrangler not only the code was analyzed, but also tests were performed. Testing the development workflow, automated tests are performed using GitHub workflows to verify that all required node libraries and dependencies can be installed, and code can be compiled. To test the functionality of Jayvee Data Wrangler, several CSV files were imported using Jayvee Data Wrangler. It was checked if the metadata was recognized correctly and if the Jayvee script ran as expected (and therefore the data was

imported correctly). Of course, further testing was done to see if the data could be modified after import, including creating custom data types. To provide a simplified overview of these tests, several examples are presented.

- Importing a .csv file without comments¹: The CSV used to test the Jayvee Data Wrangler had also the CSV-extension. The metadata was recognized correctly and the import worked within a few seconds. Therefore, — as it was executed to import the data into the database — also the Jayvee file has been created correctly (also in all further tests).
- Import of a .txt file with comments²: This file contained the CSV, but also comments at the beginning. The comments were also detected and extracted to a separate file. Everything worked as expected.
- Importing of a large CSV file with comments³: This file’s encoding was not detected automatically. But after setting it manually to UTF-8, importing worked. The test took a few seconds longer as the previously tested files, because the file had more than 100,000 entries.

Other functionality of the software (such as displaying previously created projects or displaying the help section) was also tested and worked as expected.

7.3 Software Classification

The Jayvee Data Wrangler (because it is an open source software) is mainly oriented towards the free data wrangling tools. It performs the basic data wrangling tasks stated in chapter 2. Additionally to this, it can be used without programming experience or presupposed knowledge about Jayvee or other ETL languages. Being the first tool for data wrangling based on the Jayvee ETL language, obviously makes it a prototype. Thus, it might present some issues a user would possibly encounter while using Jayvee (e.g. execution stuck on big CSV files). Compared to other software, the Jayvee Data Wrangler stands out due to the fact that it can be used by non-programmers and is at the same time a tool to simplify the creation of ETL pipelines. Not only the processed data is exported, but also the created ETL pipeline. Table 7.2 provides a comparative overview of the features offered by the Jayvee Data Wrangler in contrast to other software. It can be observed that certain software applications necessitate prior programming knowledge. This is especially true for Python Pandas (as it is a

¹<https://geo.sv.rostock.de/download/opendata/museen/museen.csv>

²https://data.bsh.de/OpenData/DOD/MO_H_CHLA/MO_H_CHLA_2019.txt

³<https://www.stats.govt.nz/assets/Uploads/International-trade/International-trade-June-2023-quarter/Download-data/overseas-trade-indexes-june-2023-quarter-provisional.csv>

programming language library), but also to a certain degree for Apache Spark and MS Data Wrangler. Apache Spark has an easy-to-understand SQL language that allows it to be used without any programming knowledge, nevertheless, to use its whole potential programming is indeed required. MS Data Wrangler can be used without any programming experience, but it runs within a code editor/development environment, so it presents a certain barrier to pass first. Since it generates Python Pandas code, it can also be expanded to create complex evaluations, as well as automations. The Jayvee Data Wrangler, as it is a prototype, currently only supports CSV files. The software has been developed to load, filter and prepare data for other applications (for example complex evaluations). As this tool generates ETL pipelines using the Jayvee language, the user can also update the data and create automations by running the Jayvee script (in certain time periods).

Software/Library	Programming required	Supports other files than CSV	Generate statistics	Generate Evaluation	Create ETL pipelines or automations
Python Pandas	yes	yes	yes	yes	yes
Microsoft Data Wrangler	partially	yes	yes	partially	partially
Apache Spark	partially	yes	yes	yes	yes
Amazon Sagemaker Data Wrangler	no	yes	yes	yes	yes
Alteryx Analytics Cloud Platform	no	yes	yes	yes	yes
OpenRefine	no	yes	yes	no	no
Jayvee Data Wrangler	no	no	yes	no	yes

Table 7.2: Features of the Jayvee Data Wrangler in comparison to other software.

8 Conclusion and Outlook

The objective of this thesis was to develop a data wrangling software based on the DSL Jayvee. The language components were analyzed and explained. Other software was presented and its features were taken into consideration for the requirements of the Jayvee Data Wrangler, as well as potential users of the software. At first, the software was developed as a command-line prototype and then converted to an executable Electron application. The software recreates a Jayvee ETL pipeline by guiding the user through that process. Most of the necessary data is recognized automatically by analyzing the CSV file provided by the user via a URL. Most of the required data is automatically detected by parsing the CSV file provided by the user via a URL (and if not recognized automatically encoding, delimiter and enclosing). The editing also works within the graphical user interface. The processes of deleting rows and columns, the change of value types, the creation of constraints and value types follow the rules of Jayvee which are recreated in order to visualize changes before being inserted into the Jayvee code files, which is used to permanently change the database.

Compared to other software, the Jayvee Data Wrangler presents some advantages and disadvantages that have already been shown in chapter 7. On the one hand, it stands out because it does not require any programming knowledge to use, and it also creates an ETL pipeline. On the other hand, it currently only supports CSV files passed through a URL and it cannot create evaluations yet.

The Jayvee Data Wrangler does not utilize Jayvee during the modification of the data until the user decides to permanently save the database, because of the in chapter 6 already clarified time-consuming disadvantages. If Jayvee adds incremental execution and caching in a future release, the algorithms that represent changes to the data could be rewritten, as they would likely be shorter. In future versions of the Jayvee Data Wrangler, importing other files from the internet like XLSX or importing files from the file system could be implemented. If more file types are supported in future releases of Jayvee, they could also be integrated into Jayvee by extending the functions that detect the metadata. Therefore, within the corresponding files on GitHub, annotations are made that explain adding further features.

Another example of extending the Jayvee Data Wrangler is the possibility of adding support for other database systems, like PostgreSQL. The feature of generating statistics can also be extended. As generating statistics is not yet part of Jayvee, only some basic statistics were developed. More complex statistics — which could be generated not only column by column, but over many columns — may be implemented in the future. Even a feature to automatically detect abnormalities in the dataset and generate evaluations could be developed. In the current version of the Jayvee Data Wrangler (0.1), the data inside the database can be modified by editing the database using other software. In future versions, this feature should be integrated into the software. Aside from that, editing parameters like the maximal execution time of Jayvee files or the project folders' location should be editable within the Jayvee Data Wrangler in a future version. Also, as Jayvee is updated regularly, its feature extensions can find their way into the Jayvee Data Wrangler. Hopefully, other programmers will contribute and suggest further features, or develop their own software based on this project, because that is why the Jayvee Datta Wrangler is published open source.

Overall, this thesis has created an open source software that can (semi)automatically perform data wrangling tasks via a user interface. The software also brings Jayvee to a larger audience as it cuts the need of understanding Jayvee, as well as it speeds-up the creation of Jayvee scrips.

Appendices

A Screenshots of the Software

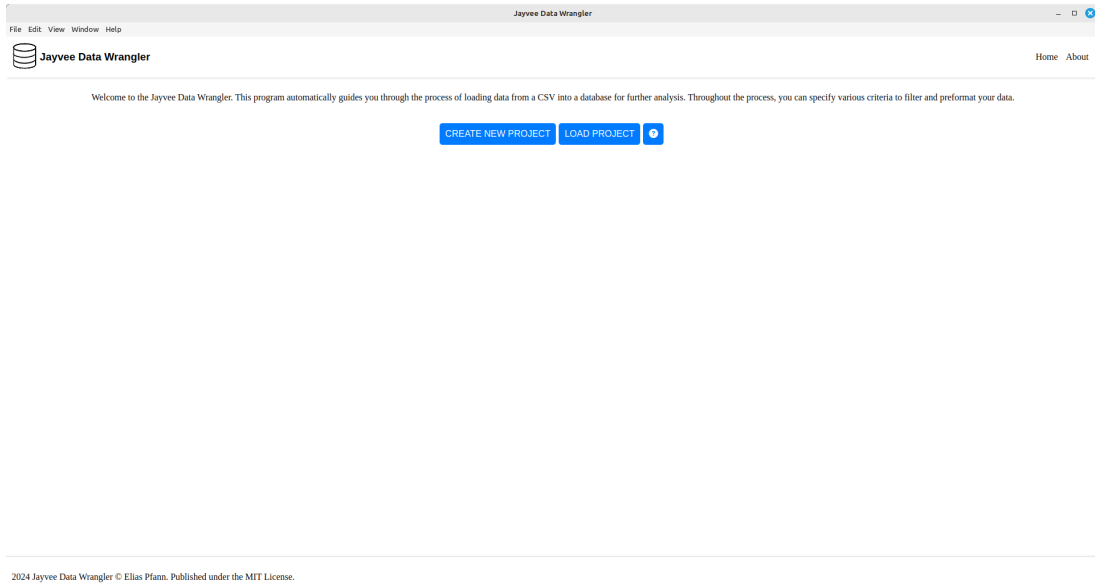


Figure A.1: The entry page of the Jayvee Data Wrangler. Users can create a new project, load previously created projects, or display help/instructions.

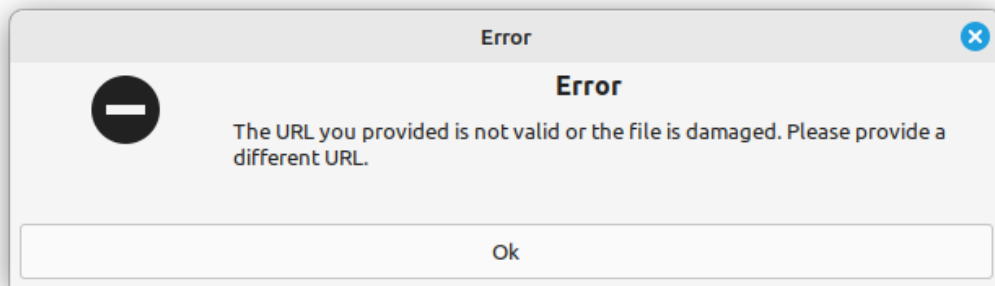
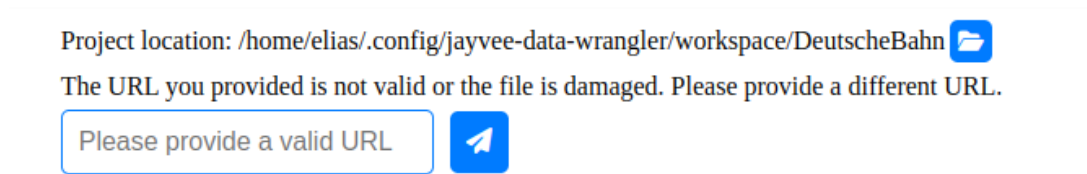


Figure A.2: Screenshot of the Jayvee Data Wrangler showing a displayed warning caused by in this case an invalid URL.

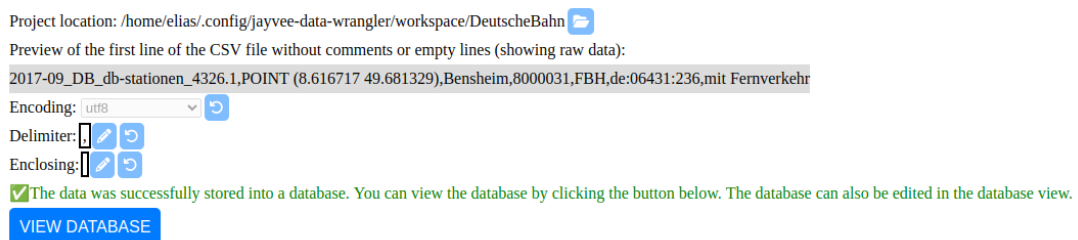


Figure A.3: Screenshot of the Jayvee Data Wrangler showing a successful import of a CSV into a database.

Create new constraint

You can create a new constraint type to filter or restrict values of columns. More about constraints can be read in the help section.

Enter the name of the constraint (You can use A-Z, a-z and 0-9):

Choose the base valuetype for the new constraint:

Which kind of constraint do you want to create to append it to one or more columns?

Forbidden values separated by comma (e.g. ms, ns):

CREATE CONSTRAINT

Figure A.4: Screenshot of the Jayvee Data Wrangler showing the creation of a constraint.

Create new valuetype

You can create a new valuetype with constraints. The constraints are used to filter or restrict the values of the columns. More about constraints can be read in the help section.

Enter the name of the new valuetype (Allowed characters are A-Z and a-z):

Choose the base valuetype for the new valuetype:


Choose a constraint for the new valuetype:

ADD ANOTHER CONSTRAINT





CREATE VALUTETYPE

























Figure A.5: Screenshot of the Jayvee Data Wrangler showing the creation of a value type.

Appendix A: Screenshots of the Software

 **Jayvee Data Wrangler**

Search:





CREATE VALUETYPE
CREATE CONSTRAINT
VIEW VALUETYPES AND CONSTRAINTS
Row deleted!

	FID	geom	Station	Haltestellennummer	Betriebsstelle	Haltestellenschlüssel	Verkehr	
Valuetypes	text	text	text	integer	text	text	text	
Actions	 	 	 	 	 	 	 	
21	2017-09_DB_db-stationen_4326.21	POINT (8.665351 49.553302)	Weinheim(Bergstr)	8000377	RWE	de:08226:4109	mit Fernverkehr	
22	2017-09_DB_db-stationen_4326.22	POINT (9.392754 49.42149)	Adelsheim-Nord	8000423	RADN	de:08225:6777	nur Regionalverkehr	
23	2017-09_DB_db-stationen_4326.23	POINT (9.395127 49.403123)	Adelsheim Ost	8000424	TAD	de:08225:6398	nur Regionalverkehr	
24	2017-09_DB_db-stationen_4326.24	POINT (8.624464 49.21847)	Albersweiler(Pfalz)	8000466	RAR	de:07337:8053	nur Regionalverkehr	
25	2017-09_DB_db-stationen_4326.25	POINT (8.450413 49.688881)	Biblis	8000503	FBL	de:06431:270	nur Regionalverkehr	
26	2017-09_DB_db-stationen_4326.26	POINT (7.966149 49.205386)	Annweiler am Trifels	8000582	RAN	de:07337:8054	nur Regionalverkehr	
27	2017-09_DB_db-stationen_4326.27	POINT (8.161602 49.57622)	Asselheim	8000625	RASH	de:07332:1003	nur Regionalverkehr	
28	2017-09_DB_db-stationen_4326.28	POINT (8.580125 49.44867)	Neu-Edingen/Friedrichsfeld	8000631	RMF	de:08222:2331	mit Fernverkehr	
29	2017-09_DB_db-stationen_4326.29	POINT (8.602787 49.193299)	Bad Bergzabern	8000691	RBZB	de:07337:4561	nur Regionalverkehr	
30	2017-09_DB_db-stationen_4326.31	POINT (8.170701 49.460633)	Bad Dürkheim	8000698	RBDH	de:07332:116	nur Regionalverkehr	

Showing 21 to 30 of 203 entries


















Figure A.6: Screenshot of the Jayvee Data Wrangler showing the deletion of multiple rows. This leads to cutting them out visually before storing the changes in the database via a click on the save button by the user.

Jayvee Data Wrangler

Search:

CREATE VALUETYPE CREATE CONSTRAINT VIEW VALUETYPES AND CONSTRAINTS If you edit anything, you can only use following characters: A-Z, a-z and 0-9

	FID	geom	Station	Haltestellennummer	Betriebsstelle	Haltestellenschlüssel	Verkehr	
Valuetypes	text	text	text	text	text	text	FilterVerkehr	
Actions								
1	2017-09_DB_db-stationen_4326.1	POINT (8.616717 49.681329)	Bensheim	8000031	FBH	de:06431:236	mit Fernverkehr	
2	2017-09_DB_db-stationen_4326.2	POINT (9.338469 49.441157)	Seckach	8000042	RSE	de:08225:8203	nur Regionalverkehr	
3	2017-09_DB_db-stationen_4326.3	POINT (8.167962 49.564384)	Grünstadt	8000137	RGR	de:07332:1009	nur Regionalverkehr	
4	2017-09_DB_db-stationen_4326.4	POINT (8.675442 49.403567)	Heidelberg Hbf	8000156	RH	de:08221:1160	mit Fernverkehr	
5	2017-09_DB_db-stationen_4326.5	POINT (8.126204 49.197741)	Landau(Pfalz)Hbf	8000216	RLA	de:07313:8304	nur Regionalverkehr	
6	2017-09_DB_db-stationen_4326.6	POINT (8.433402 49.477987)	Ludwigshafen(Rh)Hbf	8000236	"RL	RL T"	de:07314:2080	
7	2017-09_DB_db-stationen_4326.7	POINT (8.468921 49.470254)	Mannheim Hbf	8000244	RM	de:08222:2417	mit Fernverkehr	
8	2017-09_DB_db-stationen_4326.8	POINT (8.273146 49.045678)	Wörth(Rhein)	8000254	RWRT	de:07334:1731	nur Regionalverkehr	
9	2017-09_DB_db-stationen_4326.9	POINT (8.356448 49.63494)	Worms Hbf	8000257	FWOR	de:07319:4415	mit Fernverkehr	
10	2017-09_DB_db-stationen_4326.10	POINT (9.113093 49.348484)	Mosbach-Neckarelz	8000264	RNZ	de:08225:9180	nur Regionalverkehr	

Showing 1 to 10 of 203 entries

« < 1 2 3 4 5 ... 21 > »

Figure A.7: Screenshot of the Jayvee Data Wrangler showing the applying of a custom value type. In this example, a value type with Base-value text and constraint Denylist was chosen to remove all entries with "mit Fernverkehr". This results in the entire row being deleted.

Constraints and valuetypes

Created Value Types

RemoveBensheim of type text with constraints: Bensheim

Created Constraints

Bensheim with base value text of type Allowlist with values: Bensheim

Figure A.8: Screenshot of the Jayvee Data Wrangler showing the listing of all created constraints and value types within a project.

Please select a project to load by clicking on the name:

- HaltestellenBayern
- HaltestellenBerlin
- HaltestellenHessen
- HaltestellenSachsen

Figure A.9: Screenshot of the Jayvee Data Wrangler showing an example of a list of created projects. By clicking on a name, the user can open the project and if they want to navigate back, by clicking on the home button in the menu.

B Interaction Between Components

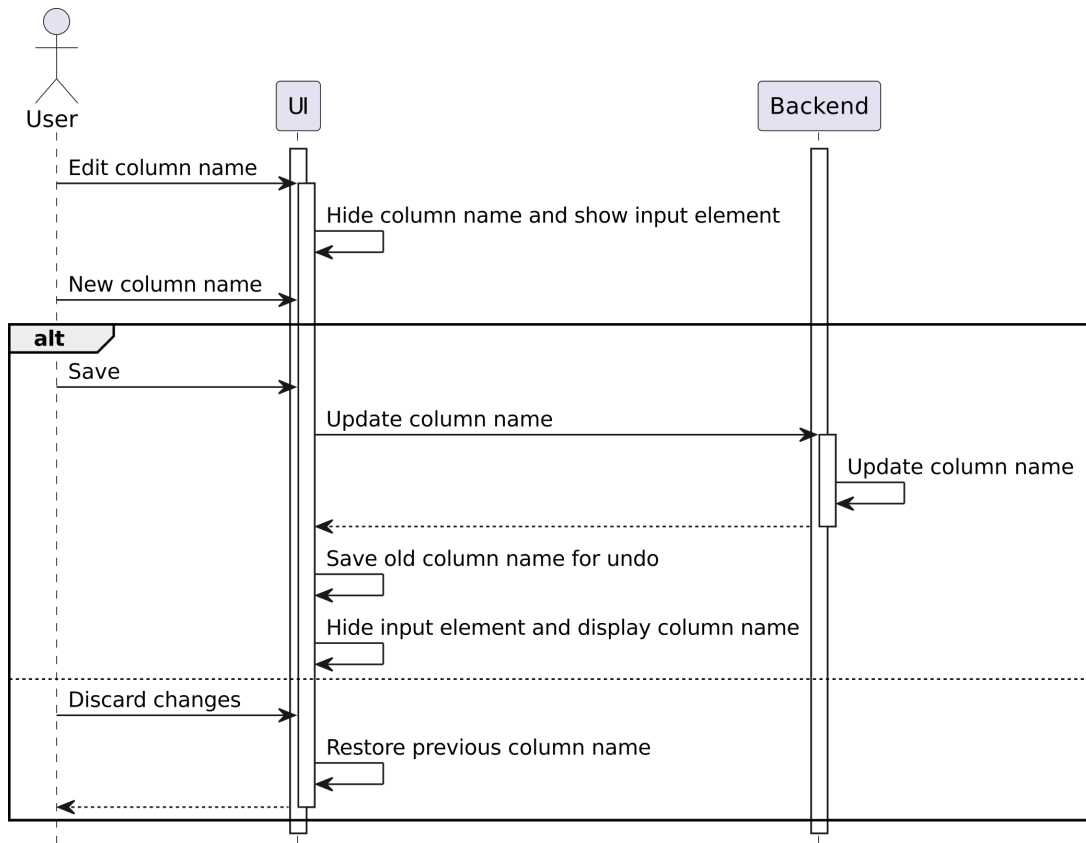


Figure B.1: Interaction between the user and Jayvee Data Wrangler while editing a column name.

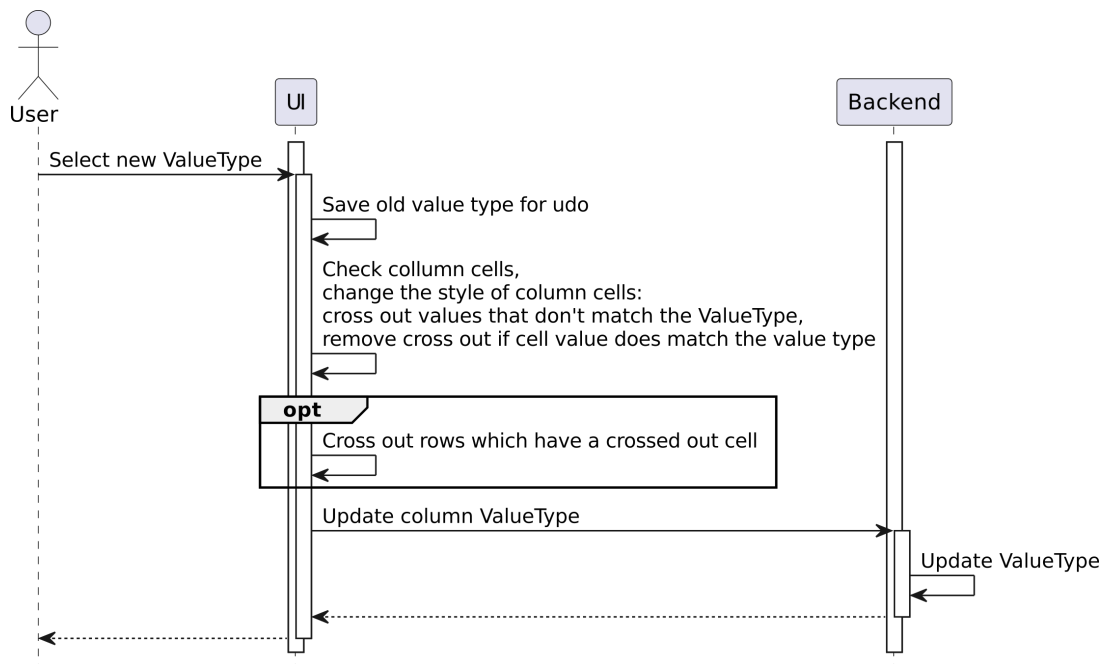


Figure B.2: Interaction between the user and Jayvee Data Wrangler while changing a value type.

Appendix B: Interaction Between Components

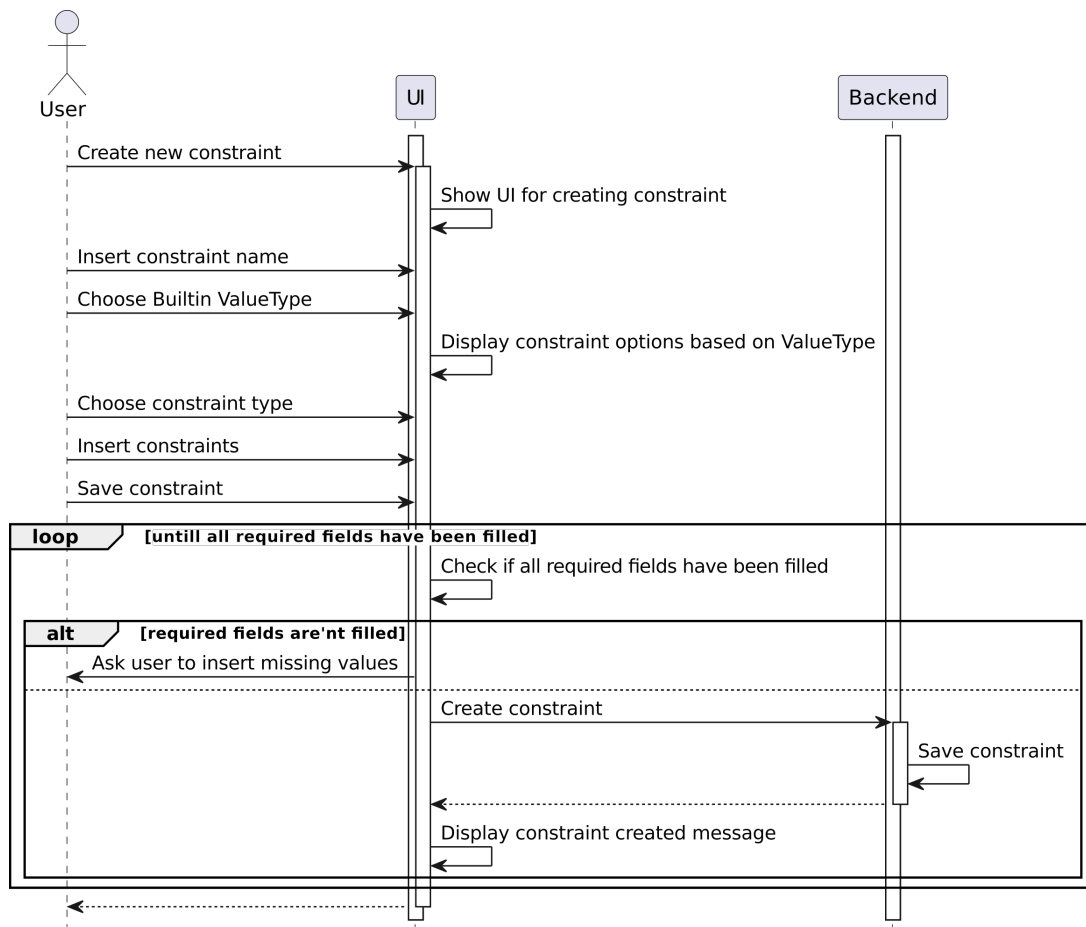


Figure B.3: Interaction between the user and Jayvee Data Wrangler while creating a new constraint.

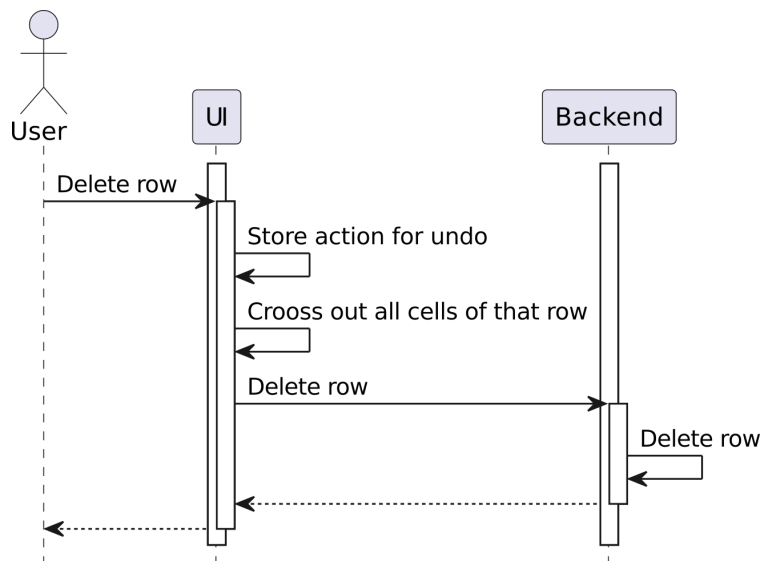


Figure B.4: Interaction between the user and Jayvee Data Wrangler while deleting a row.

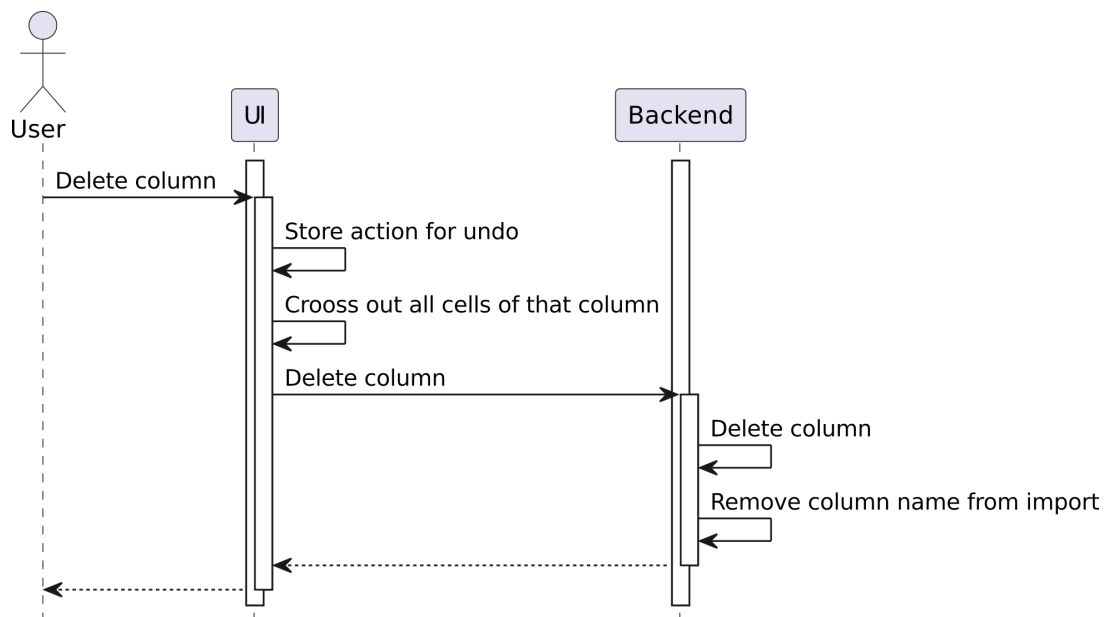


Figure B.5: Interaction between the user and Jayvee Data Wrangler while deleting a column.

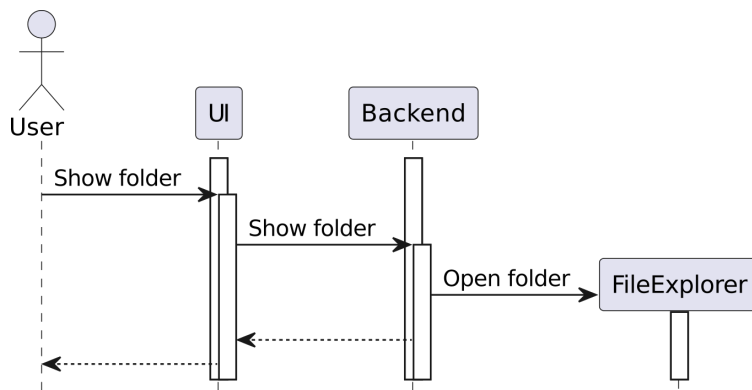


Figure B.6: Interaction between the user and Jayvee Data Wrangler while displaying the folder that contains the database, Jayvee Script and CSV file.

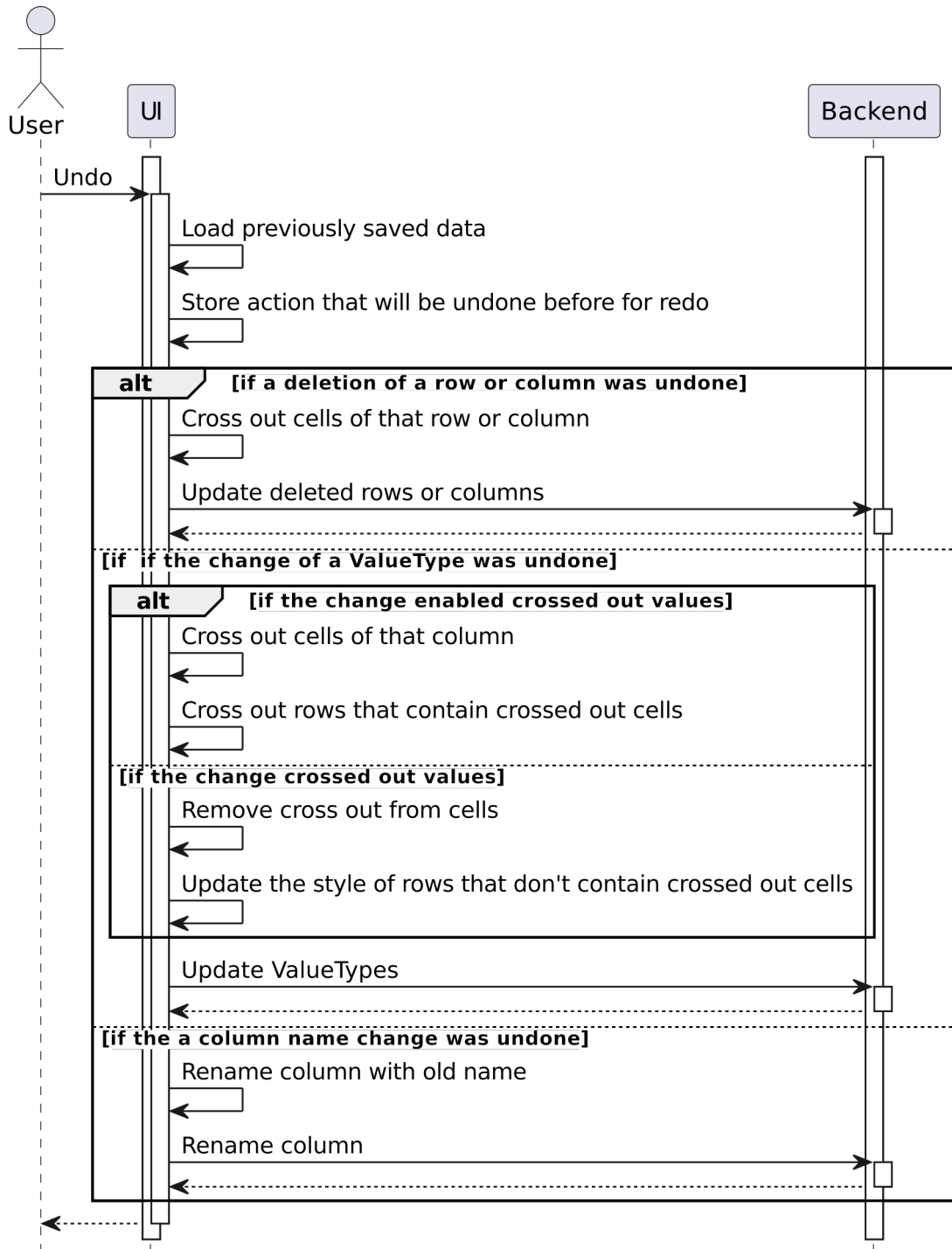


Figure B.7: Interaction between the user and Jayvee Data Wrangler while undoing the last action. Exclusively the renaming of columns, deleting rows or columns and changing the value type can be undone.

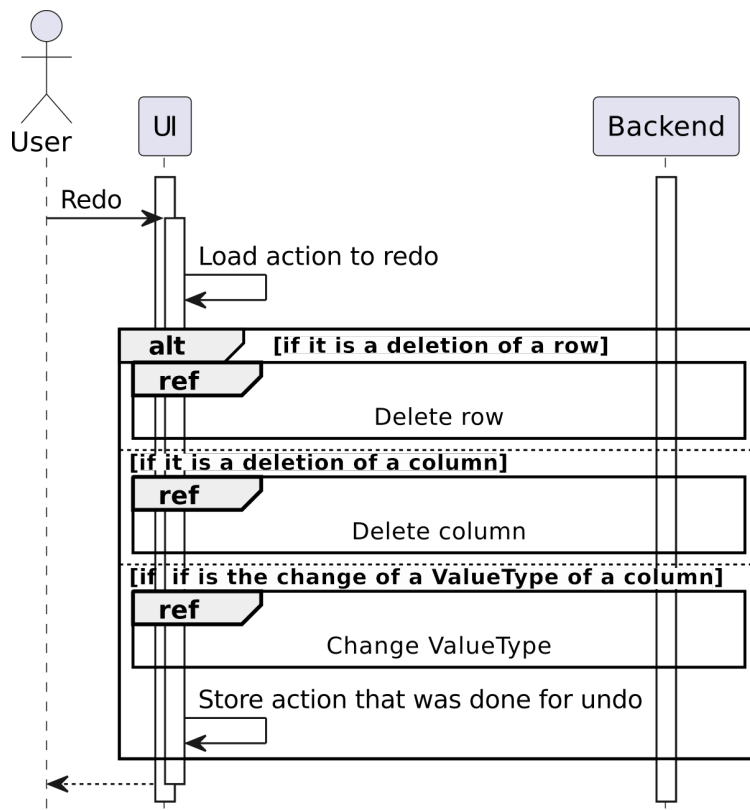


Figure B.8: Interaction between the user and Jayvee Data Wrangler while redoing the last action. Exclusively the renaming of columns, deleting rows or columns and changing the value type can be redone.

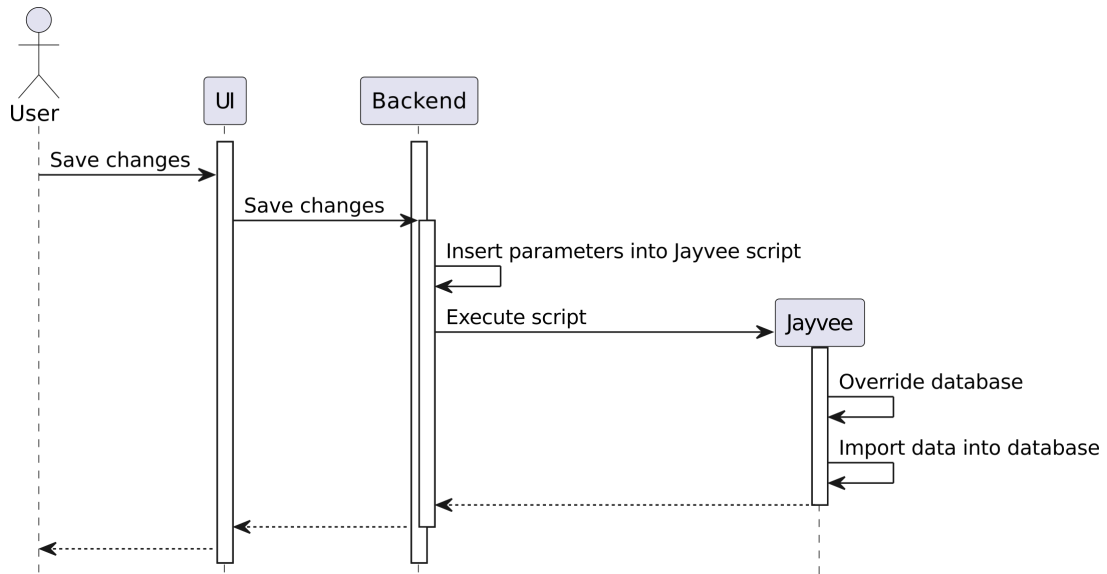


Figure B.9: Interaction between the user and Jayvee Data Wrangler to save changes. Updating of the database by executing it is also shown.

C Components and Algorithms of the Jayvee Data Wrangler

```

1 import { contextBridge, ipcRenderer } from 'electron';
2 import * as createHTMLElements from './src/helpers/createHTMLElements.js';
3
4 contextBridge.exposeInMainWorld('electron', {
5   getPath: (name: string) => ipcRenderer.invoke('getPath', name),
6   getDirname: () => __dirname,
7   createDirectory: (dirName: string)
8     => ipcRenderer.invoke('createDirectory', dirName),
9   send: (channel: string, data: any) => ipcRenderer.send(channel, data),
10  invoke: (channel: string, ...args: any[])
11    => ipcRenderer.invoke(channel, ...args),
12  on: (channel: string, func: (...args: any[]) => void) =>
13    ipcRenderer.on(channel, (event, ...args) => func(...args)),
14  once: (channel: string, func: (...args: any[]) => void) =>
15    ipcRenderer.once(channel, (event, ...args) => func(...args)),
16  removeListener: (channel: string, func: (...args: any[]) => void) =>
17    ipcRenderer.removeListener(channel, func),
18  sendSync: (channel: string, data: any)
19    => ipcRenderer.sendSync(channel, data),
20  showErrorDialog: (message: string)
21    => ipcRenderer.send('show-error-dialog', message),
22  createHTMLElements: createHTMLElements,
23 });

```

Listing C.1: Content of the preload file used to expose functions securely privileged into the renderer process.

```

1 let pipeline: { [key: string]: any } = {
2   directory: "",
3   fileName: "",
4   url: "",
5   commentLines: 0,
6   encoding: "",
7   delimiter: "",
8   enclosing: "",
9   rowsToDelete: "",
10  colsToDelete: "",
11  header: [] as string[],
12  valueTypes: [] as string[],
13  databaseType: "",
14  database: "",
15  table: "",
16  databasePath: "",
17  createdValueTypes: [] as Array<any[]>,
18  createdConstraints: [] as Array<any[]>,
19 };

```

Listing C.2: Pipeline object to store data that will be inserted into the Jayvee file.


```

1  /**
2  * Creates a folder inside the workspace directory.
3  * If the folder already exists, it throws an error.
4  * @param {string} dirName – The name of the folder to be created.
5  */
6  export async function createDirectory(dirName: string): Promise<string> {
7      const userDataPath = app.getPath("userData");
8      const newFolderPath = path.join(userDataPath, "workspace", dirName);
9      console.log("Creating directory: " + newFolderPath);
10     if (!fs.existsSync(newFolderPath)) {
11         fs.mkdirSync(newFolderPath, { recursive: true });
12         console.log("Directory created successfully");
13         return newFolderPath;
14     } else {
15         throw new Error("Directory already exists.");
16     }
17 }

```

Listing C.3: Creation of the project folder within the workspace in the user directory.

```

1  // Routing to view database
2  ipcMain.on("viewDatabase", (event, [databasePath, tableName]) => {
3      mainWindow?.loadURL(
4          `file://${path.join(__dirname, "src", "viewDatabase", "viewDatabase.html")}?databasePath=${encodeURIComponent(databasePath)}&tableName=${encodeURIComponent(tableName)}`
5      );
6  });

```

Listing C.4: Function that loads *viewDatabase.html* into the main window and hands over the database path and table name.

```

1  let dataTable = $('#database').DataTable({
2      dom: 'f<"toolbar">rtip', // Add a "toolbar" class to the dom
3      // Add the columns using the header loaded from the database and add
4      // a column for the delete buttons
5      columns: [ '', ...columns, { title: '',
6                  defaultContent: deleteRowButton.outerHTML } ],
7      width: '100%',
8      scrollCollapse: true,
9      fixedHeader: true,
10     paging: true,
11     searching: true,
12     columnDefs: [{
13         targets: '_all',
14         orderable: false,
15     }],
16     order: [],
17     // This function handles the display of entries
18     // if the user navigates to another page
19     drawCallback: function () {
20         [...]
21     }
22 });
23 // Add a row below the header with the ValueTypes
24 valuetypes = await window.electron.invoke('getValuetypes');
25 // Create a new row for the valuetypes
26 let valueTypeRow = document.createElement('tr');
27 valueTypeRow.innerHTML = '<td>Valuetypes</td>';
28 columns.forEach((col: any, index: number) => {
29     const uniqueID = `dropdown-${index}`;
30     const dropdown = htmlHelpers.createDropdown(uniqueID, allowedValuetypes,
31         valuetypes[index], false, false, null, null);
32     const td = document.createElement('td');
33     td.appendChild(dropdown);
34     valueTypeRow.appendChild(td);
35 });
36 // Append the row to the table header
37 document.querySelector('#database thead')?.appendChild(valueTypeRow);
38 columns.forEach((col: any, index: number) => {
39     const dropdown =
40     document.getElementById(`dropdown-${index}`) as HTMLSelectElement;
41     if (dropdown) {
42         // Handle dropdown value switch
43         dropdown.addEventListener('change', () => {
44             const uniqueID = `dropdown-${index}`;
45             saveDropdown(uniqueID, 'saveValueType')
46             // Activate the save button when the dropdown value changes
47             saveInputButton.classList.remove('disabled');
48             undoButton.classList.remove('disabled');
49             // Update the rows if the value doesn't fit the type anymore
50             checkValueType(dropdown, index);
51             modifyDisplay(dropdown, index);
52         });
53     }
54 });

```

Listing C.5: Creating a datatable to view the database. This involves loading the header and ValueTypes. Afterwards the data is added.

```
1 // Add the rows to the datatable
2 window.electron.on('dbEachRow', (row) => {
3   const rowData = Object.values(row);
4   // Add the row data to the DataTable
5   const rowNode = dataTable.row.add(rowData).draw(false).node();
6   // Store the row ID as a data attribute on the row
7   $(rowNode).attr('data-row-id', row.rowid);
8   rowCount++;
9   // Draw the DataTable after every N rows
10  if (rowCount % drawAfterRows === 0) {
11    dataTable.draw(false);
12  }
13 });
```

Listing C.6: Adding data to the data table



References

- Alteryx, Inc. (2024a). *The alteryx approach to generative ai for analytics* [Available at <https://www.alteryx.com/wp-content/uploads/media/whitepaper/alteryx-approach-to-generative-ai-for-analytics-whitepaper-en.pdf>, last visited on 2024-05-22]. Alteryx, Inc.
- Alteryx, Inc. (2024b). *Alteryx platform* [Available at <https://www.alteryx.com/products/alteryx-platform>, last visited on 2024-05-22]. Alteryx, Inc.
- Alteryx, Inc. (2024c). *Data sources* [Available at <https://help.alteryx.com/current/en/designer/data-sources.html#idm44990421136480>, last visited on 2024-05-22]. Alteryx, Inc.
- Amazon Web Services Inc. (2024a). *Data wrangling tool - amazon SageMaker data wrangler - AWS* [Available at <https://aws.amazon.com/sagemaker/data-wrangler/>, last visited on 2024-05-22]. Amazon Web Services, Inc.
- Amazon Web Services Inc. (2024b). *Prepare ML data with amazon SageMaker data wrangler - amazon SageMaker* [Available at <https://docs.aws.amazon.com/sagemaker/latest/dg/data-wrangler.htm>, last visited on 2024-05-22]. Amazon Web Services, Inc.
- Apache Software Foundation. (2024). *Apache spark™ - unified engine for large-scale data analytics* [Available at <https://spark.apache.org/>, last visited on 2024-05-22]. Apache Software Foundation.
- AWS Professional Service. (2024). *Aws wrangler: Pandas on AWS*. (Version 3.7.3) [Available at <https://pypi.org/project/awswrangler/>, last visited on 2024-05-22].
- Endel, F., & Piringer, H. (2015). *Data wrangling: Making data useful again* [Available at <https://www.sciencedirect.com/science/article/pii/S2405896315001986>, last visited on 2024-05-20]. *IFAC-PapersOnLine*, 48(1), 111–112.
- Furche, T., Gottlob, G., Libkin, L., Orsi, G., & Paton, N. (2016). *Data wrangling for big data: Challenges and opportunities* [Available at <https://openproceedings.org/2016/conf/edbt/paper-94.pdf>, last visited on 2024-05-20].

- GitHub, Inc. (2024). *The legal side of open source* [Open source guides] [Available at <https://opensource.guide/legal/>, last visited on 2024-05-27]. GitHub, Inc.
- Lockyer, D. (2022). *Node SQLite3 API* [Available at <https://github.com/TryGhost/node-sqlite3/wiki/API>, last visited on 2024-05-20].
- Mew, J. (2023). *Introducing the data wrangler extension for visual studio code* [Available at <https://devblogs.microsoft.com/python/data-wrangler-release/>, last visited on 2024-05-21]. Microsoft Corporation.
- Mozilla Foundation. (2024, January 12). *Async function - JavaScript | MDN* [Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function, last visited on 2024-05-19]. Mozilla Foundation.
- Murthy, A. K. (2017). *Big data analysis using hadoop and spark* [Available at <https://digitalcommons.memphis.edu/cgi/viewcontent.cgi?article=2808&context=etd>, last visited on 2024-05-22].
- NumFOCUS, Inc. (2024). *Pandas documentation — pandas 2.2.2 documentation* [Available at <https://pandas.pydata.org/docs/>, last visited on 2024-05-21]. NumFOCUS, Inc.
- OpenJS Foundation. (2023a). *Build cross-platform desktop apps with JavaScript, HTML, and CSS | electron* [Available at <https://electronjs.org/>, last visited on 2024-18-05]. OpenJS Foundation and Electron contributors.
- OpenJS Foundation. (2023b). *Introduction | electron* [Available at <https://electronjs.org/docs/latest/>, last visited on 2024-05-18]. OpenJS Foundation and Electron contributors.
- OpenRefine. (2024). *OpenRefine* [Available at <https://openrefine.org/>, last visited on 2024-05-22]. OpenRefine.
- OpenRefine developers. (2024). *OpenRefine user manual* [Available at <https://openrefine.org/docs>, last visited on 2024-05-29]. OpenRefine.
- Rupp, C., & SOPHISTen, d. (2020). *Requirements-engineering und -management* (7., aktualisierte und erweiterte Auflage). Carl Hanser Verlag GmbH & Co. KG.
- Shafranovich, Y. (2005). *Common format and MIME type for comma-separated values (CSV) files* (Request for Comments RFC 4180) [Available at <https://datatracker.ietf.org/doc/rfc4180>, last visited on 2024-05-06]. Internet Engineering Task Force.
- The JValue Project. (2024). *User docs* [Available at <https://jvalue.github.io/jayvee/docs/user/intro/>, last visited on 2024-17-03]. Professorship for Open-Source Software.

- Voleti, R. (2020). *Data wrangling- a goliath of data industry* [Available at <https://www.ijert.org/research/data-wrangling-a-goliath-of-data-industry-IJERTV9IS080122.pdf>, last visited on 2024-05-20]. *International Journal of Engineering Research and*, V9(8), 273–276.