ZURICH UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

Using spreadsheet documents as requirements specifications for automatic software generation

Submitted by:
MARTIN FRICK
15-725-161

Supervisor:
DR. MICHAEL WAHLER
Advisor:
DR. PROF HANS PETER-HUTTER

Institute of Information Technology (InIT)
MSE Computer Science

January 31, 2024

ZURICH UNIVERSITY OF APPLIED SCIENCES

Abstract

Institute of Information Technology (InIT)

MSE Computer Science

Using spreadsheet documents as requirements specifications for automatic software generation

by MARTIN FRICK

We propose an approach to generate applications from spreadsheets. This includes that the spreadsheet formulas and data tables are available in the generated application and can be worked with. The idea is to be able to use spreadsheet documents as requirements specifications for automatic software generation. There are existing approaches that do the same. However, such approaches depend on the spreadsheet which causes bad performance and limited concurrent access options. Further, there are low-code platforms available which allow to generate applications based on spreadsheets but fail when it comes to making use of existing formulas within the spreadsheet. We elaborated a new approach that is fast and can work independently from the spreadsheet after an initial import. Also, it allows to make use of formulas already defined in the spreadsheet. We call this new approach the "headless spreadsheet approach", as it uses the concept of headless spreadsheets which provide spreadsheet functionalities without GUI as in traditional spreadsheet software. The approach consists of two stages. At the first stage, a spreadsheet is imported to an application. The data from the spreadsheet is then transferred to an attached relational database. At the second stage, the user of the application is presented with the formulas as defined in the imported spreadsheet. The formulas can be evaluated upon request with the help of a headless spreadsheet instance which is created based on the data from the database. Correctness tests have shown that our solution is robust. Moreover, performance tests have shown that our approach is significantly faster than existing solutions which interact with a traditional spreadsheet. Also, the tests have shown that the approach is suited to be used by multiple users concurrently.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Requirements engineering in software development is difficult because it involves knowing and specifying all the requirements in advance. Also, there are areas where it is challenging to specify a problem. In such areas, tools are often used to model the problem.

One possibility for such tools which can be used to model requirements are spreadsheets. They are often worked within an organisation, but they are usually not used for software generation. Still, they are used to support important processes in organisations, and they serve as the basis for many business decisions. There are various applications for spreadsheets, including inventory management, educational purposes, finance and scientific modelling.

On the other hand, many organisations experience considerable delays in being able to update their software to reflect changes to their software requirements. Therefore, such companies are too slow to take advantage of trends and may miss business opportunities.

Since important parts of business logic are defined in spreadsheets, which are currently not used in software development, we see great potential here to reduce the time-to-market of software. Thus, we want to show a way of exploiting the potential of spreadsheets in companies by using spreadsheet documents as requirement specifications for automatic software generation.

Software generated from spreadsheet documents makes it possible to react quickly to changing requirements. This is due to the fact that they can be adapted by domain experts who are close to daily business. As the number of professional spreadsheet users is estimated to be many times larger than that of professional software developers [16], it would be of great benefit if we could harness this largely untapped potential of the spreadsheets maintained by this community for software development.

As spreadsheets are commonly used in many organisations, adaption would involve less effort and thus, it could accelerate Model-driven Development (MDD) practices. In this context, domain experts such as product managers capture requirements in spreadsheets which serve as models. On the other hand, software developers offer transformations to convert these models into executable code automatically.

Following the MDD approach, we provide a transformation from spreadsheets to executable code. This transformation can be used by domain experts to generate software from a spreadsheet which captures defined requirements automatically.

1.2 Related work

The use of spreadsheet documents as a means of specifying requirements for automatic software generation is not a new concept. In this section, we will review some of the related works on this topic.

1.2.1 Interacting with spreadsheets

Previous work has shown how spreadsheet document formulas can be usable [18]. The corresponding approach uses a web service to write inputs into a predefined sheet, trigger calculations inside the spreadsheet document based on the inputs given and read the outputs from another predefined sheet. Hereinafter we refer to the corresponding approach as the "spreadsheet approach". In the spreadsheet approach, the web service renders information to the user according to what is defined in the spreadsheet documents. It further allows the user to give as many inputs as needed and to receive the appropriate outputs. For this to work, an "Inputs" and "Outputs" sheet need to be manually added and mapped to the existing business logic in the spreadsheet document. This can be done to any arbitrary document.

However, the corresponding solution interacts with the spreadsheet and relies on the spreadsheet software as a formula calculation engine. In this case, the spreadsheet has to be accessed for each operation. It serves as a database, which causes poor performance. The reason is that spreadsheets have to be fully loaded with all their components when accessed. They are primarily designed for working with it manually and not for programmatic interactions. Performing frequent read and write operations, especially in a large dataset, can thus lead to performance issues. Accordingly, to improve performance, there is a need to find a different approach so that it is not required to access the spreadsheet anymore when working with the data.

1.2.2 Business platforms

Furthermore, several business tools have recognised the use of spreadsheets for generating software from them and the potential benefits it offers.

An example of such tools are Enterprise Low-Code Application Platforms (LCAPs) [11]. They offer the possibility to create applications for which data from different sources, including spreadsheet documents, can be imported into the connected database. LCAPs enable it that users without programming skills can create applications using visual tools and do not necessarily need to use software code. This has the advantage that domain experts without programming knowledge can prototype their applications on their own and be more efficient. Thus, allowing them to create customised solutions, contributing to an accelerated software development life cycle.

There are countless LCAPs available on the market. One that focuses specifically on spreadsheets is Softr [34]. It allows users to extract data from spreadsheets and integrate it into applications. This can be, for example, databases, tables or lists stored in Google Sheets or Microsoft Excel. Within the application, users can access this data and use it to create workflows based on spreadsheet data visually. Moreover, users can create forms, dashboards or user interfaces that access and interact with spreadsheet data. With the created applications, the data from spreadsheets can be accessed and edited without modifying the spreadsheet itself.

Analytics and business intelligence (ABI) platforms [10] are another example of tools to generate applications based on spreadsheets. ABI platforms are designed to help make sense of big data. These platforms can be used to explore data, create interactive visualisations and gain deeper insights into business operations.

Similar to LCAPs, common functions of ABI platforms include data integration, data modelling, data visualisation, and data analytics. ABI platforms also allow it to use spreadsheets as the basis for generating the application, which has a database with the imported data attached.

It becomes clear that LCAPs and ABI platforms have very much in common. The main difference observed in practice between the tools is that ABI platforms typically have a stronger focus on supporting decision-making with the help of advanced analytics, e.g. predictive modelling and artificial intelligence.

It can be concluded that LCAPs, as well as ABI platforms, can use spreadsheet documents to accelerate application development, automate business processes, create custom applications and generate reports based on spreadsheet data. This enables more efficient use of spreadsheet data in modern applications and workflows. Both allow it to build applications with relational databases generated from spreadsheets. The users further have various possibilities to work with the data. Although they offer the possibility of creating new calculations with the imported data, they do not use any formulas already contained in the spreadsheets. Furthermore, when using such a platform, users cannot access the source code, i.e., they cannot change their application programmatically.

1.3 Problem statement

Given that spreadsheets are widely used by domain experts, who outnumber professional software developers, and are used to capture business logic, there is a potential for automating the generation of software by using spreadsheets as requirements specification. To make this possible, a suitable approach must be found.

The spreadsheet approach can be used for this task. The according solutions use the spreadsheet as a database and formula calculation engine. However, in that case, the spreadsheet needs to be loaded for every operation, which causes bad performance.

While existing business tools like LCAPs and ABI platforms acknowledge the use of spreadsheets for generating software that works independently of the imported spreadsheet, they fall short in harnessing existing spreadsheet formulas and providing programmatic access to the generated applications.

It would be beneficial to have a tool that can make use of formulas already defined in the spreadsheet and does not depend on it anymore after an initial import. Therefore, it is required to have the spreadsheet data available in a database instead of having to access the data within the spreadsheet. This allows for independence from the spreadsheet. Consequently, data modifications can be performed on the database instead of on the spreadsheet. This offers a great benefit, as the spreadsheet does not have to be loaded anymore to be accessed as in the spreadsheet approach. There can further be taken advantage of the use of performance optimisations provided by a Database Management Systems (DBMS). It is expected that the operations on the database can be performed within a few milliseconds. The data from the database can subsequently be used for evaluating the spreadsheet formulas. Working with the data from a database instead of in a spreadsheet also simplifies collaboration, as a DBMS is designed to handle concurrent operations efficiently.

Our research aims to address this gap by proposing a new methodology that works as described. The methodology should allow the automatic generation of software from spreadsheet documents, specifically transforming them into web applications. Web applications as they are easily accessible and allow for real-time collaboration. This enables the use of spreadsheets to define requirements for the automatic generation of software with the help of data tables and formulas.

The primary objectives include transferring spreadsheet tables to a relational database as well as evaluating formulas on the database-stored data within the generated software. This way, we have an application independent of the original spreadsheet and can perform operations on the attached database, which can be done faster than when having to access the spreadsheet for every operation.

1.4 Research questions

We apply constructive research to create a methodology allowing the use of spreadsheet documents as requirements specifications for automatic software generation. More specifically, in the generated software, we want the tables from spreadsheets available in an attached relational database. It should then be possible to calculate formulas contained in the spreadsheets within the generated software without needing to access the original spreadsheet but with the data transferred to the database. Consequently, the generated web application can work independently from the spreadsheet.

After a corresponding methodology has been elaborated, we provide an implementation of the proposed methodology that serves as a showcase to demonstrate how it could work out in practice. The showcase further outlines the scope of applicability of the methodology.

The provided implementation further allows us to test it for correctness and performance. The tests are performed to assess our solution's correctness and to test whether it offers better performance than the spreadsheet approach which depends on interaction with the spreadsheet. It is necessary to answer these questions; otherwise, the provided solution would not provide any value.

The problems which need to be solved for this are summarised in the research questions below:

- Q1: How can tables in a spreadsheet document be identified and transferred to a relational database?
- Q2: How can the formulas contained in a spreadsheet document be evaluated on the data in the database?
- Q3: Is the correctness of the elaborated methodology ensured?
- Q4: How does the performance of the found solution compare to the spreadsheet approach?

In the following, we answer each of the research questions. By doing so, we ensure that our proposed methodology allows to use spreadsheet documents as requirements specifications for automatic software generation. We further prove that our solution works as intended by performing correctness testing. We also conduct performance tests to determine whether our proposed methodology can deliver better results than the existing spreadsheet approach.

1.5 Objective and Structure

To provide a background to the topic, in Chapter 2, theoretical foundations and technologies related to the research questions to be answered are given. Therefore, the definition of the term model-driven development is given. Also, the difference between models at runtime and models at compile time is outlined and explained how a programme can interact with spreadsheets in both scenarios and where the differences are. In this context we also introduce the concept of headless spreadsheets. Further, spreadsheet tables are analysed in order to provide the necessary knowledge about their structure to transfer them to a relational database. It is also necessary to look at types of spreadsheet formulas and cell references in order to make use of them. When it comes to analysing spreadsheet formulas, Abstract syntax trees are a concept that must be explored as well. To finally understand how the formulas are processed in our generated software, we examine the concept of dependency graphs.

After the theoretical foundation is laid, a methodology to generate software from spreadsheet documents is elaborated. We want to seize not only tabular data in spreadsheets but also formulas. For this purpose, we outline an overview of our methodology in Chapter 3. The methodology consists of two stages.

In Chapter 4 we elaborate an approach to correctly detect and recognize tables in spreadsheet documents and transfer them into a relational database. Thus, this Chapter provides the answer to our first research question. This marks the first stage of our methodology and combines different existing technologies and methods.

The second stage of this methodology is outlined in Chapter 5 and provides a solution for how spreadsheet formulas can be made usable for our use case. It provides an answer to our second research question.

Further, in Chapter 6, a showcase is presented to demonstrate how the methodology could be used in practice and define the scope of applicability of our provided implementation. The corresponding implementation details are provided in chapter 7.

Moreover, the methodology is tested for correctness (Chapter 8) and performance (Chapter 9). Both are crucial as incorrect or bad-performing methodologies do not provide any value in practice. The Chapters provide answers to the third and fourth research questions.

Finally, in Chapter 10, limitations and future work of our methodology are outlined, whereas in Chapter 11, a summary is given.

2. Background

The following Chapter serves as a background for the elaborated methodology which allows us to use spreadsheet documents as requirements specifications for automatic software generation.

2.1 Model-driven Development

In the following, the definitions given in the book by Brambilla et al. [5] are used to describe the model-driven environment. A model can generally be seen as a simpler representation of a more complex thing. It helps to better understand how something works in a simple and understandable way. The mechanism on which it is based is abstraction, where things are reduced to their essential characteristics, while other characteristics are not represented. There often exist different levels of abstraction, for each of them other characteristics are essential. Different levels of abstraction are suitable for people with different backgrounds and different domains. These levels consist of models of other models, also called metamodels.

Models can be created by using modelling languages. A modelling language consists of representation elements which follow a specific syntax. The syntax defines elements and how each of them can be put together. Moreover, semantics needs to be defined, which gives meaning to all the elements and their connections. Consequently, semantics must also be defined for transformations of objects in a model. Thus, there needs to be a translation from the model to code. On the other hand, the semantics can also be defined in the runtime environment. The process of using a modelling language to create a model is called modelling. Being in accordance with the syntax is a prerequisite for the correct interpretation of the model created. This can be best achieved through the usage of a modelling tool [5].

Modelling languages are divided into domain-specific languages (DSLs) and general-purpose modelling languages (GPMLs).

On the one hand, DSLs are defined for a specific domain to create models that are worked with in that particular domain, while in other domains, they might not be applicable. Spreadsheet documents can be considered as DSL as they can be used to model requirements for a specific domain as finance or tracking of expenses [5].

On the other hand, GPMLs are modelling languages that can be applied to any domain. An example of this is Unified Modeling Language (UML). For example, it can be used in software engineering to model systems from different perspectives. With UML, there can different diagrams be modelled such as class diagrams or use case diagrams. The different diagrams can be separated into static diagrams, which represent the current structure of a system, and into dynamic diagrams, which define system behaviour. For example, an UML class diagram is a static diagram used as a representation of all classes of a system and helps to better understand which classes exist and how they are connected [5].

In a Model-driven Development (MDD), software engineering the focus is on models as artefacts from which to start. MDD allows it to automate the software life cycle from requirements to finished applications by using model-driven techniques. These techniques are expected to improve efficiency and reduce costs in all software engineering tasks. They further allow discussion among members by capturing and organizing knowledge of the system. Also, they allow early assessment of whether a system meets its requirements by enabling evaluation of different designs [5].

In MDD, models are traditionally used to generate software from them, which corresponds to the concept of models at compile time. In addition to this view, there are also approaches to using models during the runtime of a programme, which we refer to as models at runtime. In both cases, models are at the required level of abstraction and are a representation of a domain application [5].

2.2 Models at runtime and models at compile time

In MDD, models at runtime are used by other systems to access their data and functionality. During runtime, they can be used to make decisions using information that is available at runtime but is unknown at compile time [4]. Only during runtime, the model becomes part of the running programme. They allow it to reason and adapt the behaviour of a software system as it is being executed. This provides opportunities for making dynamic adjustments to make sure that the program performs as intended.

Models at runtime are closely connected to the systems which access them which means that any change in the runtime model affects the systems and vice versa. Thus, changing either one has a corresponding effect on the other. Further, a model at runtime can provide a possibility for creating well designed and adaptable software. It can be a way of dealing with the complexity of a software system as it has to deal with constantly changing environmental conditions that are not fully known during development [4].

An example of a model at runtime in this context is Model-Driven Runtime Verification [40]. It allows checking a system's behaviour against its model to ensure to meet the specified requirements. At runtime, the verification system might use a model to monitor the system's execution and detect deviations from the expected behaviour. This allows it to early detect errors.

Models at compile time, on the other hand, refer to representations of software models during program compilation. In other words, it describes how the model is created and used during the compilation processes, before the program is executed, i.e. before the runtime. Models at compile time aim to generate software code from models, which is achieved through transformation. This enables the reuse of models and the adaptability to different environments [5].

An example of a model at compile time is a UML class diagram transformed into code. UML, in this case, is the modelling language used to form a model. For each element in the UML class diagram, it needs to be defined how it can be translated into code. The code might, in the end further be completed manually.

2.3 Spreadsheets

A spreadsheet is a software application that allows users to organise data in tabular form. It is designed to process numerical data, text as well as formulas and allows the user the ability to perform calculations, generate visual representations like charts, and produce reports from the data entered into its cells.

The basic structure of a spreadsheet is a grid of cells arranged in rows and columns. Each cell can contain data, such as numbers, text, dates or formulas. This allows users to input information, perform mathematical operations, and leverage predefined functions for complex calculations.

Microsoft Excel, Google Sheets and LibreOffice are popular examples of spreadsheet software widely used for personal and business purposes. There are many applications for spreadsheets, including inventory management, educational purposes and finance. Spreadsheets are used to support important processes in organisations, and they serve as the basis for many business decisions. Spreadsheets are extensively used across different industries. An analysis of computer use across 95 organisations in different industries and continents in 2007 has shown that spreadsheets accounted for 7.4% of the total computing time of professional workers [35].

When starting with MDD, it may further be necessary to learn new modelling languages, such as a low-code platform modelling language. This might be an obstacle for a practitioner to get involved. Even when the upfront training needed to learn a new modelling language is not very high, it could still be a reason which slows down the adoption of MDD practices. Therefore, it might be highly beneficial if a modelling language could be used that most of the relevant working personnel already know. While software developers often prefer programming to model UML diagrams, domain experts would first have to learn UML and probably also prefer to express themselves in a language related to their domain. Thus, a common basis is missing.

As spreadsheets are widely known and used, they could fill this gap and could potentially be leveraged as models for MDD practices in software engineering. They can be considered as a DSL and serve as a common basis as a modelling language for people with different backgrounds. Thus, enabling the use of spreadsheet documents as a requirement specification for automatic software generation. In this context, it would be beneficial to make use of not only tabular data but also formulas within the generated software.

In MDD, spreadsheets can potentially be used either as model at runtime or as model at compile time. In the following two sections 2.4 and 2.5, we outline how each approach works.

2.4 Interacting with spreadsheet documents at runtime

As we have outlined in 1.2.1 there are applications such as the spreadsheet approach that interact with spreadsheets to use them as a database and formula calculation engine. The according operations are performed at runtime. There are many tools and methods for how spreadsheets can be interacted with as models at runtime. EPPlus [8] is an example of a possible technology that allows the user to interact with a Micrsoft Excel document at runtime from a programme. It is a library for .Net applications that allows to manipulate the Excel document and to read from it. Other tools are available such as web APIs that are made available by spreadsheet providers. An example of this is Microsoft Graph [26] which allows software to access spreadsheet data over the internet. It allows it to send HTTP requests to these APIs to read or modify data in cloud-based spreadsheets.

Such technologies allow the user to manipulate a spreadsheet, within a short programme. Some basic operations such as opening a workbook, writing to it and reading defined cells can be performed. Further, results of calculations performed by spreadsheet formulas can be read or formulas be written to the spreadsheet.

Besides reading and writing, such tools further allow to create new spreadsheet documents from scratch. Also, they provide different formatting options for cells, such as font styles, colours and borders. Different types of charts can further be added and customised as well as populated with data. It can be noted that with such tools it is possible to do many of the operations, that can performed within the spreadsheet software.

Figure 2.1 illustrates what interacting with spreadsheet documents at runtime looks like. The logic which enables to access the spreadsheet with the help of EPPlus or another technology is usually included in a dedicated service which is utilised by the user to perform operations such as evaluating formulas and manipulating the data. The spreadsheet itself is not necessarily part of the application but is a component which is used at runtime.

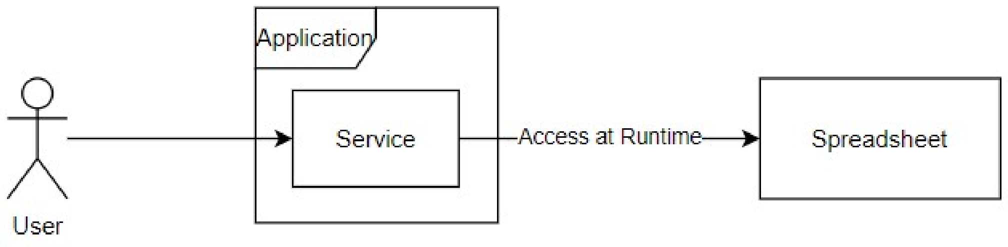


FIGURE 2.1: Interacting with spreadsheet documents at runtime

Although the described technologies offer a simple solution and decent performance, the performance might not be sufficient for certain applications where there is a high number of requests and spreadsheets containing a large amount of data. This is due to the fact that such technologies use spreadsheet documents as models at runtime and subsequently fully load the entire file for each request. In the respective use case, the spreadsheet document serves as a database which causes performance loss. Moreover, unlike a DBMS, spreadsheets cannot guarantee data integrity constraints as well as a possible security risk to be used as a database by applications. Data consistency can further not be ensured when multi-user access is required.

2.5 Interacting with spreadsheet documents at compile time

Applications that interact with spreadsheets documents as models at runtime offer bad performance. The performance could possibly be increased by interacting with them as models at compile time. However, most technologies available interact with spreadsheet documents at runtime. There are fewer options available when it comes to using spreadsheets at compile time. So there must a suitable approach be developed. Accordingly, there must be transformation rules set which define how spreadsheets can be transformed into code such that it can be further processed within the software.

For spreadsheets, the transformation into code is not a difficult task, as each sheet can be considered as two-dimensional array. For example, a spreadsheet document can be considered a dictionary with a key-value pair for every sheet. While, the keys could represent the sheet names, the respective values could contain the sheet content as a two-dimensional array. An example of how this could be done is given below.

```
const sheets = {
 'Sheet1': [
   ['2', '6', '1'],
   ['', '', '=SUM(1, 3)'],
   ['=Sheet2!$A3', '5', ''],
  ],
 'Sheet2': [
   ['', '2', '=Sheet1!$C1'],
   ['', '5', '=SUM(9, 3)'],
   ['=Sheet1!$C1', '3', ''],
  ],
};
```

Still, after the spreadsheet has been transformed into a dictionary, it is required to further make use of the generated code, especially when it comes to evaluating formulas. There are different possibilities for this task. One possibility is the concept of headless spreadsheets which is outlined in Chapter 2.6.

Moreover, when generating an application from a spreadsheet, it might make sense to transfer spreadsheet tables to a relational database attached to the application for the reasons of data consistency, data persistence and data security. Also, If we could store this data then in a database it would allow it to derive the spreadsheet in the form of a dictionary from the database such that we do not depend to read it again from the spreadsheet document at runtime. Thus, we are required to correctly identify tables in our spreadsheet document and transfer them to a database beforehand. The challenges of this task are described in Chapter 2.7. Accordingly, the dictionary containing the spreadsheet could be extracted from the database instead of from the spreadsheet directly. Figure 2.2 illustrates how this can look like. Following this approach, after an initial import of the spreadsheet, the application can extract the spreadsheet data from the database instead of from the spreadsheet itself. This allows it to be independent from the spreadsheet document.

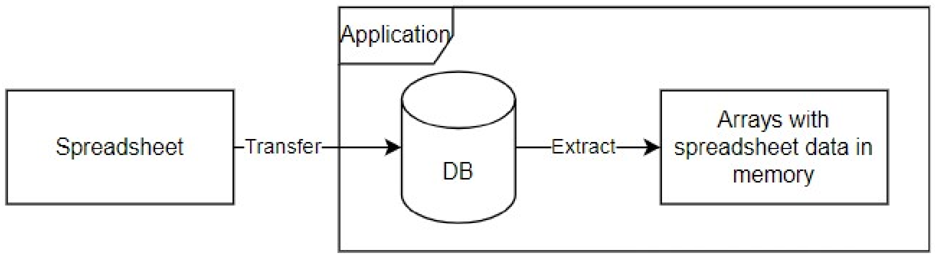


FIGURE 2.2: Making use of spreadsheet documents at compile time

2.6 Headless spreadsheets

To make further use of the spreadsheet data at compile time, the concept of headless spreadsheets can be used. Traditionally, spreadsheet applications like Microsoft Excel or Google Sheets provide a graphical user interface where users can interact with and manipulate spreadsheet data visually. However, headless spreadsheets are instances of spreadsheets that can perform spreadsheet operations without a visible user interface and without having to load the entire spreadsheet when performing operations. They can work "headlessly" in the background, executing spreadsheet operations programmatically without the need for a graphical interface and have less overhead which improves performance. Thus, they allow developers to automate spreadsheet operations within their applications and achieve high performance without having to interact with the original spreadsheet document.

An example of a headless spreadsheet library is HyperFormula [13]. It allows to create an instance of headless spreadsheets based on the original spreadsheet. It requires the spreadsheet data as input to create an instance. We can take the data we extracted from the spreadsheet beforehand as outlined in section 2.5. After an instance has been created, common spreadsheet functionalities such as creating worksheets, modifying cell values, formatting data and performing calculations are possible. It includes a parser of spreadsheet formulas and allows it to evaluate them on the HyperFormula instance.

Regarding performance, HyperFormula allows efficient calculations, especially with large datasets and complex formulas. As they are designed to efficiently process calculations on a large number of cells, they are suitable for applications that require extensive data manipulation. Unlike when interacting with spreadsheet applications, headless spreadsheets often employ techniques like lazy evaluation and dependency tracking as well as minimizing unnecessary recalculations for improving performance [14].

Figure 2.3 illustrates what interacting with spreadsheet documents with the help of headless spreadsheets can look like. After we have read the data from the spreadsheet to the attached database, no interaction with the spreadsheet document is required anymore. We are able to derive the required arrays from the database which allows us to build the headless spreadsheet instance according to the original spreadsheet. Thus, we are not required to access the spreadsheet outside the application at runtime anymore.

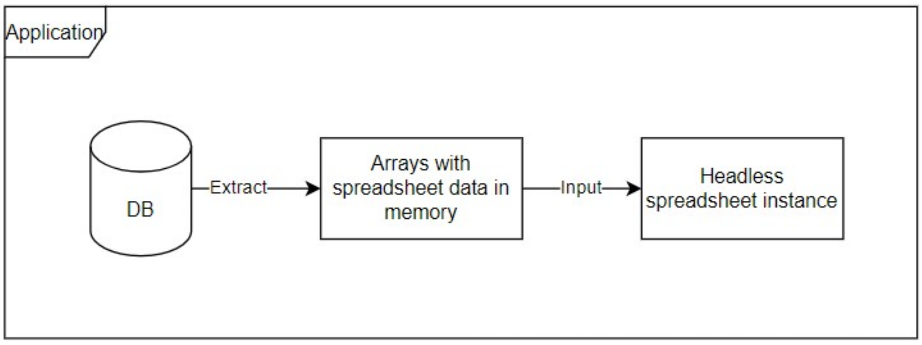

Application
DB
Extract
Arrays with spreadsheet data in memory
Input
Headless spreadsheet instance

FIGURE 2.3: Interacting with spreadsheet documents at compile time

2.7 Identifying spreadsheet tables

As we have outlined, interacting with spreadsheets at runtime offers bad performance and should be avoided. A possible solution would be to read the data contained in the spreadsheet document and transfer it to a Database Management System (e.g. Microsoft SQL Server). From there we can work with the spreadsheet data within our application. This would possibly allow for significantly better performance. However, it is necessary to find a suitable approach to solve this task which includes identifying all the tables in the document.

For this purpose, there needs first an approach to be found on how the data in the spreadsheet document can be correctly detected and recognized such that a suitable data model can be derived. Tables in spreadsheets can be arranged and formatted in different ways which makes this a rather complex task.

As we want to transfer tabular data from spreadsheets to a relational database, it is important to understand what tables are and what they are composed of. This knowledge is important to correctly identify tables in spreadsheets, which is necessary before we can think of transferring them to a database.

Tables can in general be defined as a two-dimensional data structure to organize data. They can appear in different layouts and styles. Wang [38] introduces a comprehensive model for defining table structure. Tables have three main components, according to the author: a relation describing the elements of the table, headers and optional stubs which serve as descriptions of the contents, and visual representations of the different components such as fonts, separators, and other formatting methods.

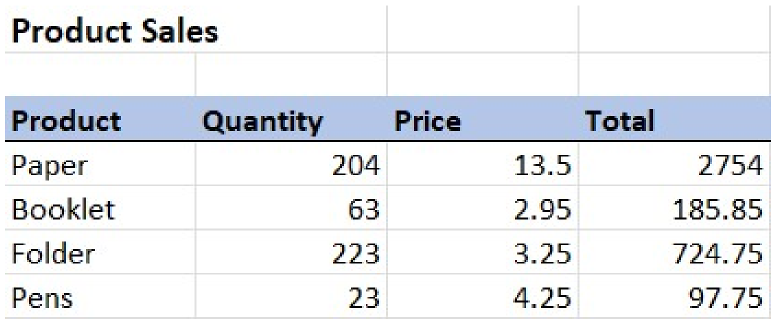
Spreadsheet tables more specifically consist of cells, i.e. the individual fields at the intersection of a row and a column. These cells can contain text, numbers or formulas that can be used to perform calculations and manipulate the data in the table. Tables can be formatted with different fonts, colours and formatting options to make them more visually appealing and easier to read.

In the context of spreadsheet tables, there exist three types of cells, namely header cells, data cells, and title cells. A header cell is a cell that provides a label that describes the content of the current column within the table. Data cells, on the other hand, contain values that are part of the body of the table and are often of the same data type within a particular column. Further, title cells contain a value that describes the content or purpose of the table.

It can also be seen that those components differ from each other with regard to their cell attributes. Cell attributes are traits of a spreadsheet cell which include font size, colour, borders, etc. Identifying the different components of the table is possible based on the attributes of the cells and their relationship to surrounding cells. The table's structure can further be determined by its separators such as borders and blank lines.

Figure 2.4 illustrates the different components that can appear in a spreadsheet table. It can be seen that it includes a title, a header row which is separated via their format and data entries. The title of the table is separated from the rest of the table with a blank line. The header row on the other hand is separated from the data entries with the help of a separating line and different background colour. This represents a common table structure. However, it must be noted that there are many more possible layouts.

When it comes to identifying tables with their different components, there are a variety of approaches in research on how to do this. They can be categorized into rule-based [7], semi-automated [3] and full automatic [17] approaches.



Product Sales

Product	Quantity	Price	Total
Paper	204	13.5	2754
Booklet	63	2.95	185.85
Folder	223	3.25	724.75
Pens	23	4.25	97.75

FIGURE 2.4: Spreadsheet table example

Rule-based approaches are based on a predefined set of rules to identify and extract tables from a spreadsheet. Data from tables is extracted by recognising the layout and formatting of the document, looking for table characteristics, and identifying those patterns. The rules are set as such that according to them a decision can be made whether a cell in the spreadsheet is a header, data entry or not relevant at all. While rule-based approaches can be effective in identifying tables with high accuracy, they can also be limited by their reliance on the predefined rules. If a document uses a non-standard layout or formatting, the rules may not be able to identify the tables within it accurately.

In contrast, semi-automatic approaches allow more flexibility in terms of the table layout as they are not fixed on a set of rules. However, semi-automatic approaches need user input to detect the table layout. With full automatic approaches on the other hand, no user input is required. Automated approaches are based on for example machine learning or data mining methods. Automatic approaches adapt to a wider range of document layouts and formats, making them more flexible for identifying tables. They can be highly effective when it comes to identifying tables, especially those with non-standard layouts and formats. However, large amounts of labelled data are needed to train the models, as well as computational power.

2.8 Spreadsheet formulas

For our methodology which aims to evaluate spreadsheet formulas, we further want to understand what components a formula consists of. When analysing spreadsheet formulas, it is important to consider that each spreadsheet software can have a different formula syntax, although they might be very similar. Also, they share a set of common functions. Still, it must be noted that for example, Microsoft Excel has some specific functions, while Google Sheets also has its own specific functions such as Google functions. For this section, we decided to focus on the formulas used in Microsoft Excel, as it is probably the most widely used spreadsheet software.

When it comes to analysing formulas in Excel, one can see that they consist of a combination of mathematical operators, functions, and cell references. They can be used to analyse data and make decisions by building complex calculations [2].

Understanding and using cell references correctly is important when making use of formulas in Excel. They can impact the accuracy of calculations and the adaptability of the spreadsheet. Formulas in Excel often reference cells or ranges of cells. By referring to cells or ranges, formulas can operate dynamically according to their contents. Formulas that refer to cells do recalculate if their value changes. Otherwise, it would be necessary to edit the formula itself if the data used in the calculation was not referenced. It is possible to reference one or more cells in a formula and thus have its contents automatically updated whenever the referenced cells change. By using A1 notation [27], it is possible to point to different sources of data using different formulas, which makes it flexible for formulas to get values.

Regarding cell references, a formula can reference a relative value, an absolute value, or a mixed value. Referencing relative rows and columns means the references are offset from the current row and column, so when copying the formula into another cell, the references may change. Excel creates relative cell references in formulas by default [2].

Absolute references, on the other hand, refer to actual cell addresses, so they do not change when the formula is copied. Absolute references have two dollar signs in their addresses, one for the row number and one for the column letter (for example, $A$1) [2].

There are aspects of both relative and absolute references in mixed references. There is only one absolute part of an address, indicated by the dollar sign, while the other part is relative [2].

Below is a table that summarises the methods of referring to cells with examples.

Type	Current sheet	Different sheet
Relative	=A1	=Sheet!A1
Absolute	=$A$1	=Sheet!$A$1
Mixed	=$A1	=Sheet!$A1

TABLE 2.1: Cell References

Besides cell references, there also exist range references. Range references are references to multiple, adjacent cells. Ranges can be specified with the colon operator (:). Similar to cell references, there exist different types of range references. Cell ranges span over multiple contiguous cells over adjacent rows and columns thus forming a rectangle. On the other hand, column and row ranges either span over an entire row or column [2].

Below is a table that summarises the methods of referring to ranges with examples.

Type	Current sheet	Different sheet
Cell range	=A1:B2	=Sheet!A1:B2
Column range	=A:B	=Sheet!A:B
Row range	=1:2	=Sheet!1:2

TABLE 2.2: Range References

There can further be multiple ranges combined. This can be done by either using the union operator (;) between two ranges. Using this operator, multiple range references can be combined into one. On the other hand, a space can be entered between two ranges to get their intersection. When this operator is used, it produces cells that are shared by both ranges [2].

When it comes to analysing Excel functions, it can be seen that there are a number of built-in functions in Excel, such as SUM, AVERAGE, and COUNT, as well as the possibility to create custom user-defined functions using Visual Basic for Applications (VBA) [2]. The most common built-in functions in Excel can be categorized into logical functions, text functions, date and time functions, mathematical functions and lookup and reference functions [15].

Logical functions evaluate logical conditions and return either true or false based on the result. Text functions manipulate text strings, such as combining two or more text strings. Date and time functions calculate dates and times, such as adding and subtracting days or calculating differences. Mathematical functions perform calculations on data in a worksheet. Lookup and reference functions are a group of functions in Excel that allow users to search for specific information in a table or list and return the corresponding value [15].

With Excel formulas, there can further be basic arithmetic calculations performed when using mathematical operators, such as addition, subtraction, multiplication, and division. Examples of arithmetic formulas are "=A1+B1", "=A1-B1", "=A1*B1", and "=A1/B1" [15].

In the table below, the different function types are summarised and for each type, there are some examples of functions of that type given.

Function type	Examples
Logical functions	=IF(A1>A2,"Yes","No") =AND(A1>B1,A1<C1) =OR(A1>B1,A1<C1)
Text functions	=CONCATENATE(A1," ",C1) =LEFT(A1,3) =LOWER(A1)
Date and time functions	=TODAY() =HOUR(A1)
Mathematical functions	=SUM(A1:C5) =SQRT(64) =ROUND(A1, 2)
Lookup and reference functions	=VLOOKUP(B4,B5:C25,2,TRUE) =INDEX(B1:B15, 2)

TABLE 2.3: Excel function types

Aivaloglou et al. have conducted research on the complexity of formulas used by Excel users [1]. More than eight million unique Excel formulas were analysed according to complexity indicators which are the number of functions and constants as well as operators. They have outlined that 18.18% of the formulas do not contain any function call or an arithmetic calculation. 51.94% of the formulas included only one function call or one arithmetic calculation, while 85.16% have two or fewer. Only 1.04% of the formulas included user-defined functions.

About 98 % of formulas use data from other cells and contain different types of references. A quarter of the formulas contain cell range references, whereas unions and intersections are infrequently used. Roughly 29% of formulas reference cells outside their current worksheet, and even 8% of formulas use references to cells in external files [1].

2.9 Abstract Syntax trees

Abstract syntax trees are an important concept in the evaluation of spreadsheet formulas. In general, an Abstract Syntax Tree (AST) is a representation of the syntactic structure found within source code. It resembles a tree-like structure. It captures the essence of the code on a higher level of abstraction, rather than considering every implementation detail. Within an AST, the source code takes the form of a hierarchical arrangement of nodes. Each node corresponds to a syntactic element such as a variable declaration or function call. These nodes are connected in a way that corresponds to their relationships within the source code [9].

ASTs can be used when it comes to analysing spreadsheet formulas [1]. In this context, a formula grammar needs to be defined that refers to a set of rules that define the syntax and structure of valid formulas in spreadsheets. It offers a formal specification that defines how the different components of a formula such as function names and operators can be combined in order to form a valid expression.

A formula grammar consists of a set of rules that define how combinations of different formula components can be made. Each rule establishes the structure and relationships between these elements, specifying both the allowed combinations and their corresponding interpretations. By defining such grammar rules, it becomes possible to parse spreadsheet formulas effectively and to generate ASTs that describe the hierarchical structure of these formulas. These ASTs allow subsequent analysis, evaluation, or transformation of the formulas [1].

To parse a spreadsheet formula to an AST, there can be different tools be used. One such tool is XLParser [39]. It can parse spreadsheet formulas and is designed to analyse spreadsheet formulas, generating compact parse trees. XLParser has a success rate of 99.99% on different data sets. However, XLParser is not very restrictive and can therefore parse formulas that a spreadsheet software would reject as invalid.

An example of an Excel formula parsed with XLParser is shown in Figure 2.5. The Excel formula which is parsed there is "SUM(A1:A25)". It can be seen that the formula is separated into different components which are further split into the smallest components of the formula which can be found on the leave nodes. The resulting AST represents the hierarchical structure of the formula, capturing the precedence of the elements such as function calls. Each element is defined as a node.

By using ASTs in the context of spreadsheet formulas, there can be gained a better understanding of the structure and semantics of formulas. This makes it easier to perform operations like formula validation and formula evaluation.

When it comes to validating formulas, ASTs serve as a tool for ensuring that spreadsheet formulas conform to the correct syntax. The construction of an AST allows it to identify syntax errors and deviations from the expected grammar. When a formula can not be parsed successfully, the AST cannot be generated. This indicates that the formula contains syntax errors.

Regarding formula analysis, ASTs are a good way to examine the formulas and their components. For example, Aivaloglou et al. [1] have used ASTs to perform research on the complexity of Excel formulas. In addition, using ASTs can be a valuable approach to finding formula dependencies. By examining the AST, it becomes possible to identify the specific cells or ranges that are referenced within a formula. This knowledge is especially useful when it comes to managing data dependencies effectively. Furthermore, studying the AST offers insights into the structure of the formula. This includes how expressions are grouped and which functions are called. Such analysis helps to understand the composition of the formula and to find formulas that can be optimised.

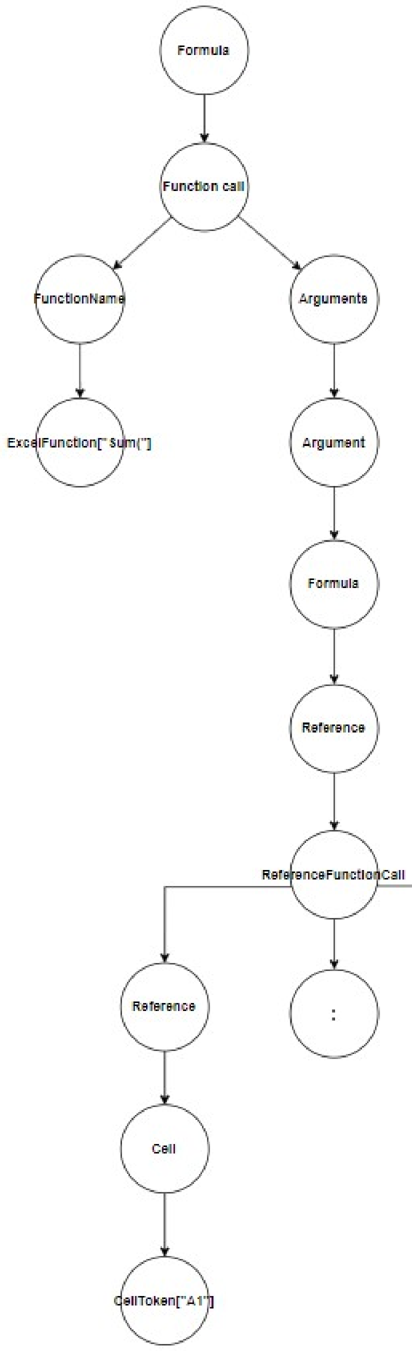


FIGURE 2.5: Ast generated by XLParser

Moreover, ASTs serve as a tool for evaluating formulas, as they encapsulate all the necessary information required to compute their results. By traversing the AST and systematically applying the defined operations and functions, it becomes possible to calculate the output of a formula. This ability to evaluate formulas proves especially valuable as formula evaluation engines. However, not all parsing tools provide this functionality. While it is for example not included in the XLParser, other tools like the Apache POI [36] provide the functionality to evaluate spreadsheet formulas based on ASTs.

2.10 Dependency graphs

Dependency graphs are a concept used to find the best order for processing cells when working with spreadsheet data. In general, a dependency graph is a graphical representation of the dependencies between different elements in a system. It is a directed graph that describes the relationships between different components. The dependency graph consists of nodes and directed edges, where the nodes represent the elements and the edges represent the dependencies between them. The dependencies indicate that one element is dependent on another to perform its function or to be executed correctly. Visualising the dependencies in a graph makes it easier to understand the relationships and determine the order in which the elements need to be processed [20].

In spreadsheets, dependency graphs can be used to visualise cells and references between them. Each cell in a spreadsheet is represented by its own node in the dependency graph. Two nodes are linked if the formula in one cell contains the address of the other cell. This concept is central to HyperFormula as it needs to understand the relationships between cells in order to find the correct order to process them. In an example formula =B1+C1, B1 and C1 must be processed before the formula can be calculated. Such an order of processing cells only exists if there is no cycle in the dependency graph. The corresponding dependency graph is shown in Figure 2.6, which shows that cell A1 depends on cells B1 and C1, which must be processed first [12].

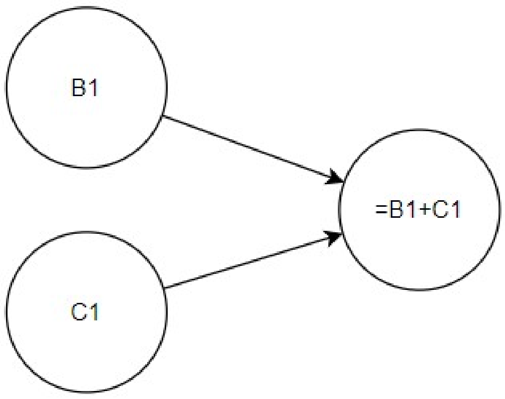


FIGURE 2.6: Dependency graph

The dependency graph can also represent ranges [12]. They are represented by their own node. It is possible to connect range nodes to cell nodes as well as range nodes to other range nodes. There can also be ranges in the dependency graph that are unused by any formula in order to improve optimisation.

In many applications, it is often necessary to utilise formulas that rely on a wide range of cells. For instance, the formula SUM(A1:A50)+B5 is dependent on 51 cells and the dependency graph must be accurately created. A significant optimisation challenge arises when multiple of such cells depend on large ranges. A scenario where multiple cells depend on similar but slightly different ranges causes a high amount of dependencies. A solution to this problem arises from the observation that the formulas can be rewritten into equivalent ones that are more compact to represent [12].

In the approach implemented by HyperFormula [12], whenever the engine encounters a range, such as A1:B10, it checks if it has already considered the range that is one row shorter, in this case, A1:B9. If it has, then it represents A1:B10 as the composition of the range A1:B9 and two cells in the last row which are A109 and B10. In general, the result of any associative operation can be obtained by performing operations on these smaller rows. There are multiple examples of such associative functions, including SUM, MAX, MIN and others. Since a range can be utilised in different formulas, it can be reused and duplicating the work during computation can be avoided. This is a particularly important consideration as working with big spreadsheets that contain many formulas is resource-intensive. Such optimisation techniques help to work with big spreadsheets and still achieve high performance.

3. Methodology overview

In the following Chapter, we outline the overview of our methodology. It consists of transferring data from the spreadsheet to a database as well as evaluating the spreadsheet's formulas based on the transferred data. This allows us to have the data within the application and not have to access the spreadsheet at runtime anymore as in the spreadsheet approach.

The transfer of data from the spreadsheet to the database marks the first stage of our methodology. It is started when a user uploads the spreadsheet to the application. After the spreadsheet has been uploaded, the functionality of the application is generated. This includes that the data from the spreadsheet is available in the database. Further, the formulas can be evaluated within the application based on the database which marks the second stage of our methodology.

For transferring the data from the spreadsheet to the database, a suitable approach is required. Tables need to be identified correctly with their components which are title, headers and data entries. It must be ensured that tables are identified and correctly transferred to the database. This part of the methodology is addressed in Chapter 4.

After the spreadsheet has been uploaded, the relevant formulas are extracted from the spreadsheet. The formulas can later be evaluated with the help of the concept of headless spreadsheets. The headless spreadsheet acts as a computational engine, evaluating formulas without the need for a graphical user interface as in a spreadsheet software. When a user triggers the evaluation, the headless spreadsheet processes the formulas, providing results in the application's user interface. Therefore, when evaluating the formulas, we have to build an instance of the headless spreadsheet that requires the spreadsheet data as input.

A spreadsheet can be represented internally as a dictionary with a key-value pair. It can be done such that the keys represent the names of the sheets, whereas the values provide the content of the sheets in the form of a two-dimensional array. This dictionary is hereinafter referred to as the "internal spreadsheet representation". We can later use this internal spreadsheet representation to build our instance of the headless spreadsheet. Thus, it serves as a fundamental building block and allows it to be further processed for formula evaluation. The internal representation needs to be generated based on the tables in the database when the evaluation is triggered by the user.

Consequently, we have to find a way to generate the internal spreadsheet representation based on the tables we have transferred to the database beforehand. The generated representation needs to resemble the original spreadsheet. In this sense, we have to transform the tables from the database back to a spreadsheet representation. Also, it must be possible to update the data and reflect the changes in the representation of the spreadsheet to guarantee that the formula evaluation is still correct as the data changes within the application. This part of the methodology is addressed in Chapter 5.

The methodology is visually represented in Figure 3.1, illustrating the flow from data transfer to formula evaluation. This visualisation serves as a guide for understanding the two stages of our methodology.

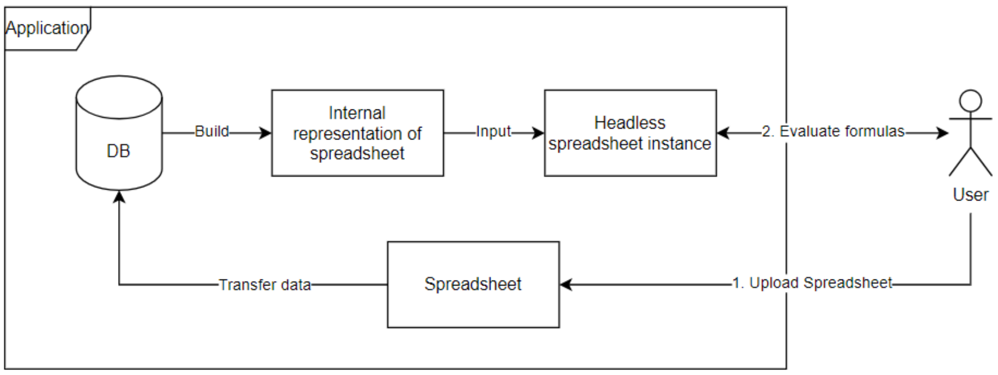


FIGURE 3.1: Methodology overview

The outlined methodology serves for using spreadsheet documents as requirements specifications. It allows to generate functionality within an application that has been previously defined within a spreadsheet. The resulting application has the spreadsheet's data available in the attached database. Formulas defined in the spreadsheet can be evaluated based on this data. Thus, the application can work independently from the original spreadsheet. As the concept of headless spreadsheets is central to our methodology, we hereinafter referred to it as the "headless spreadsheet approach".

In the field of MDD practices, the headless spreadsheet approach can be considered as hybrid approach between the model at compile time and model at runtime approach. On the one hand, the spreadsheet is translated into a dictionary, which is further used to create a headless spreadsheet at runtime.

In the subsequent Chapters, the two stages of the headless spreadsheet approach are explored deeper, providing a comprehensive understanding of the processes, challenges, and solutions.

4. Transferring spreadsheet tables to a relational database

In the following Chapter, we outline the first stage of the headless spreadsheet approach which enables us to transfer spreadsheet data to a relational database. Thus, we answer the first research question. This is an essential part of our methodology as it allows us to work with the data in the database instead of having to interact with the spreadsheet for every operation. For this purpose it is necessary to find a suitable approach to solve this task, which includes identifying all the tables in the document, deriving a data model and creating corresponding schemas in the database as well as inserting the data.

4.1 Challenges

When transferring spreadsheet tables to a relational database, we are confronted with challenges. The first challenge is to obtain the correct data model from the tables. There are technologies that can be used to read the spreadsheet document but still, it is required to correctly detect the table with its dimensions. Moreover, column headers and data entries as well as the title of a table need to be recognized accordingly. Tables in spreadsheets can have different layouts, which makes the task more difficult. We can choose between existing approaches in research for this purpose. As we have seen, there are rule-based, semi-automated and full automatic approaches to solve this challenge.

Another challenge is to obtain the right data type from spreadsheet data. In spreadsheets, cells contain different data types. An easy approach would be just to consider all the data as strings. However, our data model would be of better use if we could correctly identify the data types and consider them in our data model accordingly. Also, we must decide how to handle missing values.

There are more challenges as to detect primary keys. In research, there are existing methods to detect the primary key of a data table. For example, there are methods based on identifying functional dependencies [6] or performing an analysis [37]. Other challenges are normalisation of the data, consolidating data from different sheets and documents, exporting the data model to different formats, detecting foreign keys and more for which there also exist different approaches on how to solve them.

4.2 Overview and architecture

We present a rule-based approach to transfer tables in spreadsheet documents can be transferred to a relational database. The approach includes the following steps: table detection, table recognition, creation of a data model and storing the tables in a relational database. The architecture of the approach is given in Figure 4.1.

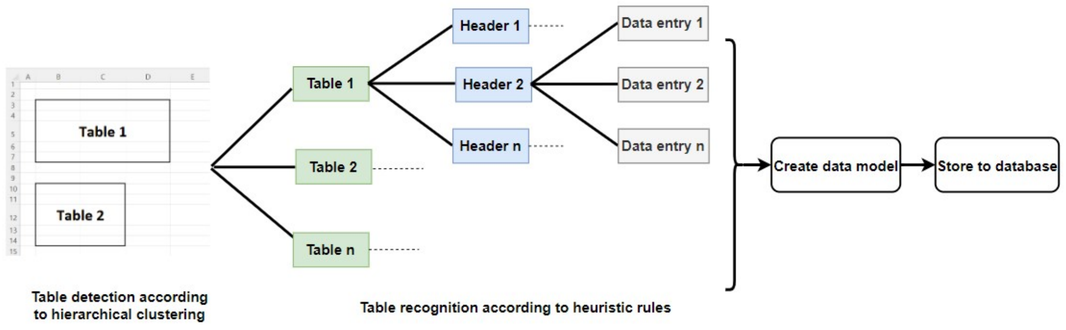


FIGURE 4.1: Architecture of proposed approach

When it comes to table detection and recognition, the following problems are common: A non-table component is incorrectly detected, a table component is incorrectly detected, an improperly assigned table component or multiple tables are labelled as one table. Therefore, the main requirement for such an algorithm is to have a high chance of correctly detecting and recognizing the tables. Only then can a useful data model be derived and the data can be correctly stored in the database.

As tables usually follow a similar structure it makes sense to use a rule-based approach for table detection and recognition. Also, they are easier to implement than more automated approaches but still can have high accuracy. Doush and Pontelli [7] propose a rule-based algorithm, which has proven well in table detection and recognition. Therefore, we decided to use their proposed algorithm in our approach. It is based on the idea that each cell should be assigned to either a title cell, header cell or data cell. The assignment of the cells is based on a set of defined rules for each of the cell types.

The algorithm includes the steps of table detection and table recognition. Table detection is the process of correctly identifying the dimensions of all tables in a spreadsheet document. On the other hand, table recognition on the other hand is the process of correctly assigning a cell to a predefined cell type. Both processes are combined within the same algorithm.

Detecting and recognizing tables correctly requires analysis of both their format and structure. As we have outlined in section 2.7, cells of a spreadsheet table can be categorised into different cell types. Further, we have outlined that those components differ from each other with regard to their cell attributes such as font size, colour, borders etc. We can use the cell attributes to decide for each cell to which cell type it should be assigned. The table's structure can thus be determined.

After tables have been correctly detected and recognized, there can be a data model created accordingly. Finally, this allows us to create the corresponding database schema and to insert the data into the database.

4.3 Table detection

In the chosen approach, the process of detecting tables is performed through hierarchical clustering, which results from categorizing the data into one of the three cell types [7]. The table detection and recognition algorithm starts from the rightmost, bottommost non-empty cell and goes through all the cells and assigns them to one of the cell types according to the defined rules. If a cell can not be assigned, it is ignored. This is done for each sheet in the spreadsheet document. A table is composed of a set of headers and each header is composed of a set of data entries. Therefore, the data is stored column-wise as shown in Figure 4.1. Organising the data this way refers to hierarchical clustering and enables the detection of multiple tables within multiple sheets and prepares the data for further processing.

The clustering hierarchy as shown in Figure 4.1 can be reflected in classes accordingly, as outlined in Figure 4.2. The hierarchy is created due to the fact that each object of a title, i.e. a table, has a field called "columns" which is a list of the related columns. Thereafter, each object of the column class has a field called "entries" which is a list of the data entries assigned to that column. Additionally, the column class includes the field "count" which assigns a number to each column which is needed for logic purposes. On the other hand, each entry has an i and a j coordinate to identify the corresponding spreadsheet cell. Multiple tables are detected on the basis that there are multiple instances of the table class.

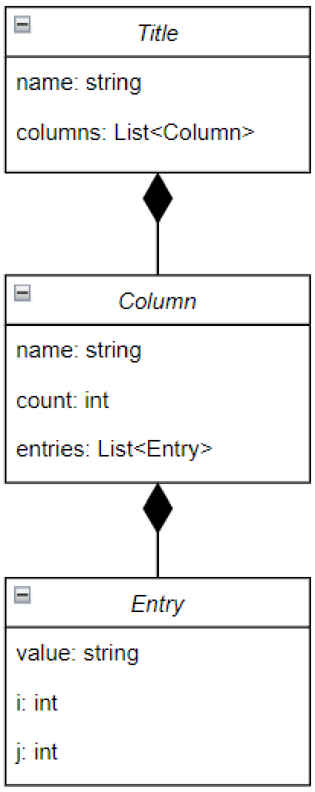


FIGURE 4.2: Class diagram of hierarchical clustering

4.4 Table recognition

Table recognition is the process of correctly assigning a cell to the according cell type. This is done with the help of a defined set of rules for each of the cell types. The rules provided by Doush and Pontelli [7] are derived from heuristic observations. The rules are followed as closely as the description in the paper allows.

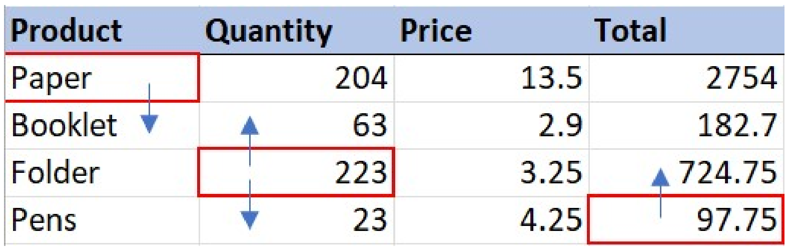
The paper proposes weighting values that can be applied to different attributes to identify the cell type. A weight value is a number that is assigned to specific cells and defines the type of cell that is attached according to the defined rules. To support the classification of cells according to one of the three types, specific thresholds are established on the weights of attributes. Each cell type is labelled based on its weight value that is equal to or exceeds its threshold. By using weighting values, it becomes easier to modify the recognition algorithm. Depending on the value of the cell type threshold, the recognition process can also be relaxed or tightened for different cell types [7]. For our approach, we give equal weight to each attribute in the rules for the three cell types so that it can be applied to different table layouts and does not weight some cell attributes over others.

In the following, the rules for correctly recognizing title cells, header cells and data cells are defined. To simplify the notation, a cell is identified according to its row i and column j. Accordingly, a cell with row i and column j can be described as C[i,j] [7].

4.4.1 Rules for data entries

The rules for data entries consider the fact that data entries forming the same row usually have the same data type and same cell attributes. So the rules apply that a cell must have neighbouring cells C[i+1, j] and C[i-1, j] which have similar formats in terms of their font size, colour etc. Also, the data types must be the same. However, there must be the fact considered that border entries, i.e., the top and bottom row only have one neighbour entry. Thus, for border entries, the check can only be applied in one direction [7].

Figure 4.3 shows an example of where the rules are applied. All the cells in the first and the last row are considered as border cells. The marked cell on the bottom right is the first cell, which is detected. As it is a bottom border cell, only the format and type of C[i+1, j] are being checked. For the marked cell on the top row, the check is performed in the other direction C[i-1, j]. As for both marked border cells, the type and format of the neighbouring cell are the same, the marked border cells are identified as data entries. If the cell, which is to be checked, is not a border cell, format and type of both, C[i+1, j] and C[i-1, j] are to be considered. In the example in Figure 4.3, this applies to the two rows, which are not border rows.



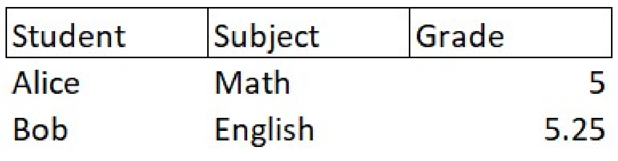
Product	Quantity	Price	Total
Paper	204	13.5	2754
Booklet	63	2.9	182.7
Folder	223	3.25	724.75
Pens	23	4.25	97.75

FIGURE 4.3: Data entries recognition example

4.4.2 Rules for headers

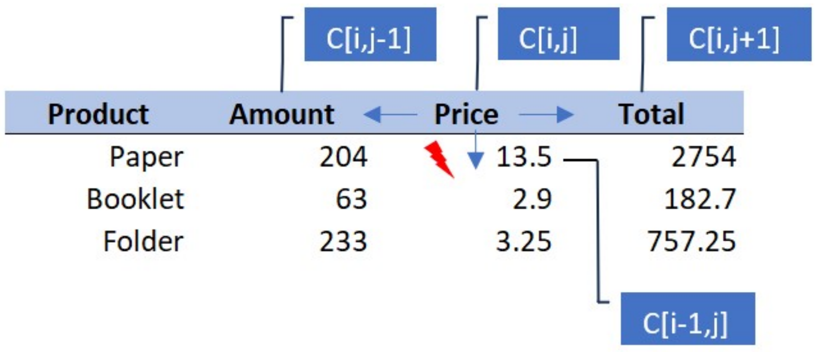
The rules for header cells are based on the observation that they have different formats than data entries, they have borders and they are equally formatted as their neighbouring header cells. Thus, a cell is identified as a header cell if one of the following rules apply [7]:

- Current cell C[i, j] is a header cell if C[i-1,j] has previously been marked as a header cell and the type of C[i,j] is string. This is due to the observation that header cells are normally of the type string because they use text to describe the category of the table column.
- Current cell C[i, j] is a header cell if C[i,j] and its neighbouring cells on the same row, C[i,j-1] and C[i,j+1] have the same format. For example, font size of C[i,j] has the same value as font size of C[i,j-1] and font size of C[i,j+1]. Further, font and background colour C[i,j] is equal to neighbouring cells C[i,j+1] and C[i,j]. Similar formatting is used in a spreadsheet to indicate a grouping of cells. Three adjacent cells in the same row that have a similar format may indicate that they belong to the same group of cells, in this case the header cells. Figure 4.5 shows how this rule is applied. Cell C[i,j] is compared to its neighbours on the same row based on its format.
- Current cell C[i, j] is a header cell if C[i,j] and its neighbouring cells C[i,j-1] and C[i,j+1] have borders. For example, C[i,j] has a left or right border or C[i,j] has a top border or a bottom border, and the same for the cells C[i,j-1] and C[i,j+1]. The bordering of adjacent cells is often an indicator of semantic information contained in the group of cells. Figure 4.4 shows an example of this rule, where the header row is identified due to borders.
- Cells C[i,j] and adjacent cells have a similar format, while C[i,j] and C[i-1,j] have not a similar format, and the type of C[i,j] is string, while the type of C[i-1,j] is not string. Also, C[i-1,j] is not an empty cell. This rule is based on the observation that header and data cells have dissimilar formats and thus can be distinguished. Another indicator might be different cell types. Where header cells must be of type string, data cells are often not. An example of this rule is shown in Figure 4.5. The current cell C[i,j] which is a header cell and the cell below it which is a data cell differ in terms of format as well as cell type, which is text for the current cell (i.e., header cell) and non-text for the cell below (i.e., data cell).
- Row i contains a cell that has already been assigned as a header cell, then cell C[i,j] is also a header cell. This rule assumes that header rows are normally displayed as a group in the same row.



Student	Subject	Grade
Alice	Math	5
Bob	English	5.25

FIGURE 4.4: Header cells recognition based on border



Product	Amount	Price	Total
Paper	204	13.5	2754
Booklet	63	2.9	182.7
Folder	233	3.25	757.25

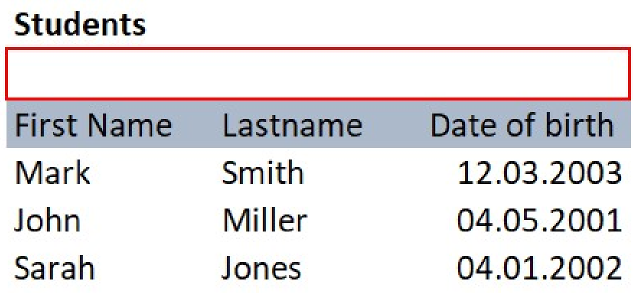
FIGURE 4.5: Header cells recognition based on format and data type

4.4.3 Rules for title cells

The rules for title cells are based on that titles are located above header cells, they are of type string and there is a set of empty rows between a title cell and header cells. Also, they usually do not have neighbouring cells. A cell is identified as a title cell if one of the following rules apply [7]:

- There has been a header cell found, C[i,j] is of type string and there are also a number of blank lines i.e. separators between header cells and title cells. This rule takes into account the observation that there are usually a number of blank lines between the title and the rest of the table.
- A title cell has already been found and in the last row scanned. This rule takes into account that subtitles are possible.
- If a header cell has been found, C[i,j-1] is empty, C[i,j+1] empty and j is the table's first column. This rule is to distinguish title cells from header cells, as for title cells, the neighbouring cells are usually empty.

Figure 4.6 shows an example of where the rules are applied. The marked row indicates the empty row between the title and the rest of the table, it serves as a separator. It can be further seen, that the neighbouring cells of the title cell, i.e., cells C[i,j-1] and C[i,j+1] are empty. Thus, the title cell can be distinguished from the headers.


Students

First Name	Lastname	Date of birth
Mark	Smith	12.03.2003
John	Miller	04.05.2001
Sarah	Jones	04.01.2002

FIGURE 4.6: Title cells recognition example

4.5 Table detection and recognition algorithm

In the following section, the algorithm proposed by Doush and Pontelli [7], which combines both, the processes of table detection and table recognition, is explained. It is used to assign each spreadsheet cell to one of the three cell types. It starts on the rightmost, bottommost, non-empty cell and applies the steps as illustrated in Figure 4.7.

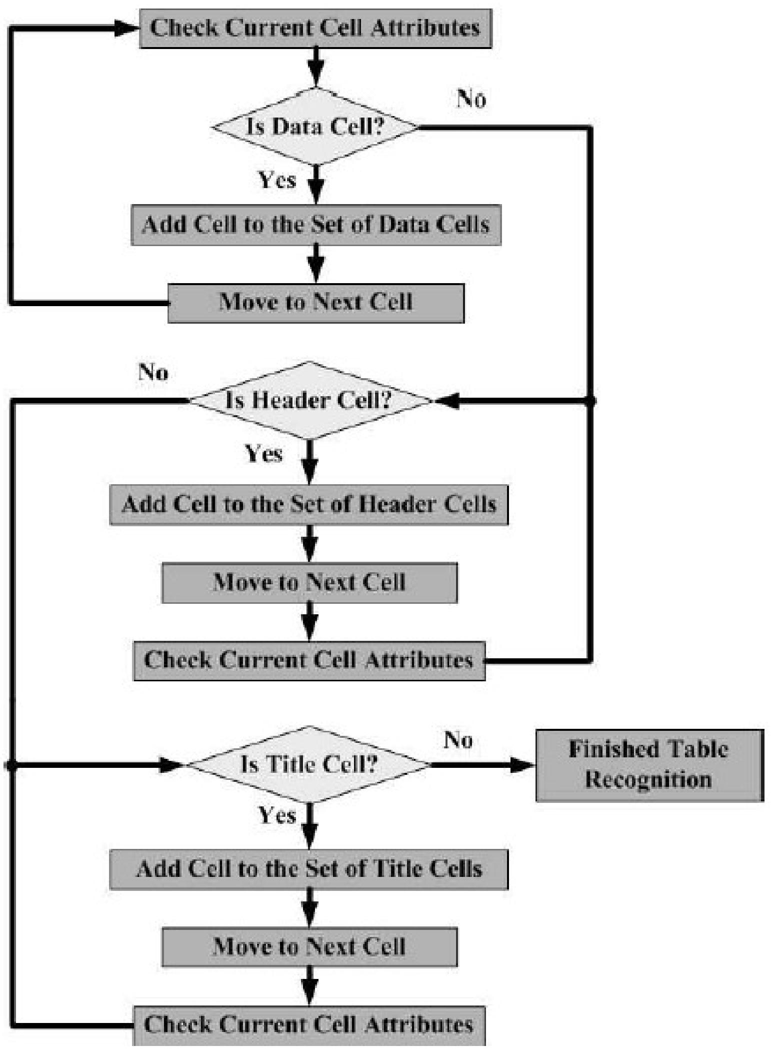

Check Current Cell Attributes
Is Data Cell?
No
Yes
Add Cell to the Set of Data Cells
Move to Next Cell
No
Is Header Cell?
Yes
Add Cell to the Set of Header Cells
Move to Next Cell
Check Current Cell Attributes
Is Title Cell?
No
Finished Table Recognition
Yes
Add Cell to the Set of Title Cells
Move to Next Cell
Check Current Cell Attributes

FIGURE 4.7: Table detection and recognition algorithm [7]

At first, the algorithm checks if the current cell is a data cell. The check is done according to the defined rule for data cells. If the current cell could be identified as data cell, an object of the data entry class is created which is added to the list of data entries and the algorithm starts from the beginning by moving to the next cell. If the current cell can not be identified as a data cell, the check for whether the cell is a header cell will be applied. This check is done according to the header cell rules. Again, if the cell can be identified as header cell, an object of the header cell class is created and added to the list of header cells as well as the algorithm moves to the next cell. Also, each new header cell gets assigned the data entries from the list which are matched to the according column. On the other hand, if a cell cannot be identified as a data cell or as a header cell, the algorithm checks if the current cell is a tile cell. If the rules apply, a new object of the title class is created and the headers detected so far are assigned to it. If the current cell is none of the three cell types, the table recognition is finished. The algorithm will start again with empty lists if there are more cells to scan. This way the whole sheet is scanned for multiple tables and the hierarchical clustering is ensured.

4.6 Creation of the data model and its storage in the database

After the table detection and recognition algorithm has completed its task, the detected tables are hierarchically clustered column by column. However, to work with the data a row-wise clustering is handier as it is natural in the human understanding of a table. Thus, there is a mapping required from the column-wise orientation to the row-wise orientation. With this mapping, it is possible to have an object for each table, whereas the headers and data entries are properties of the object. We then have one table class that contains all table data instead of having three classes with composition. This makes it easier to continue working with the data and export it to other formats. After this step is applied, the final data model is obtained from which it is more readable to export to other formats or to insert the tables into a database.

The mapping algorithm takes the hierarchical clustering and produces a new class which is the final data model. The result is illustrated in Figure 4.8. In the newly generated class, each header is stored in a string array and the data entries are stored in a two-dimensional object array.

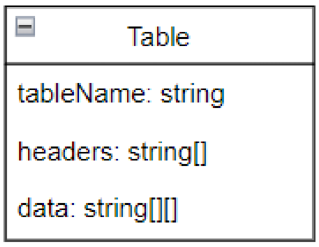
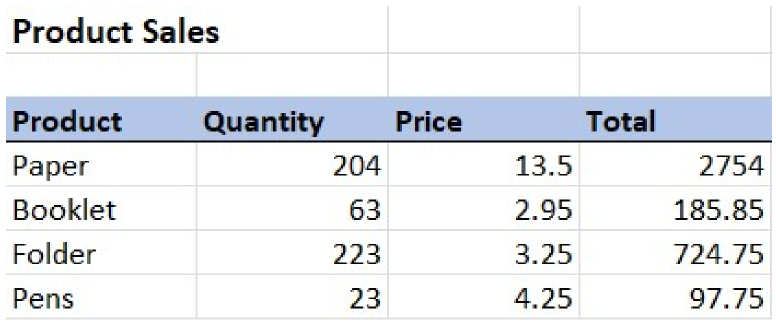


FIGURE 4.8: Table class

After the data model is obtained, the tables need to be set up in a relational database accordingly. For this purpose, a database is created, a connection to the database is established, the database schema is migrated to the database as well as the data is inserted. The table schema creation and insertion of data are processed according to the data model. This involves deriving the table schema and creating a data table with data extracted from the data model.

4.7 Example

In the following section, we apply our approach to a small example to show how it works in practice. In our example, we process a spreadsheet document containing the Product Sales table as shown in Figure 4.9. When the table detection and recognition algorithm is applied to the spreadsheet document, the class hierarchy is set up such that each spreadsheet cell is assigned to one of the cell types. Each cell belonging to a table will be assigned to one of the cell types. For each sheet in the spreadsheet document, the table detection and recognition algorithm starts by scanning the bottommost, rightmost, non-empty cell until a table is found. It can further be seen that the headers have the same format and differ from the format of data entries. Moreover, there is a separator, i.e., an empty blank line between the headers and the title. Thus, the algorithm is able to correctly identify the table. There are also no missing values that would otherwise have to be replaced by either "n/a" or "N/A".



Product Sales

Product	Quantity	Price	Total
Paper	204	13.5	2754
Booklet	63	2.95	185.85
Folder	223	3.25	724.75
Pens	23	4.25	97.75

FIGURE 4.9: Product sales table

After the algorithm has finished, the table has been detected and recognised. In our example, the algorithm detects that there is one table in the processed spreadsheet document which is the Product Sales table. It has four columns and four data entries. Whereas the "Product" column is of type String, the other columns are of type float. Further, each cell has been assigned to one of the cell types. The according hierarchy for the table is illustrated in Figure 4.10. It can be seen that the table consists of multiple columns (headers) and each column contains data entries. The data is stored column-wise. Later it is mapped to a row-wise orientation to obtain the final data model.

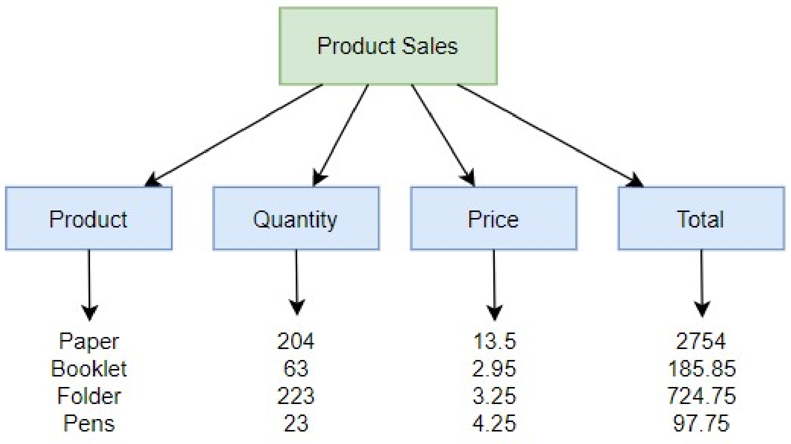
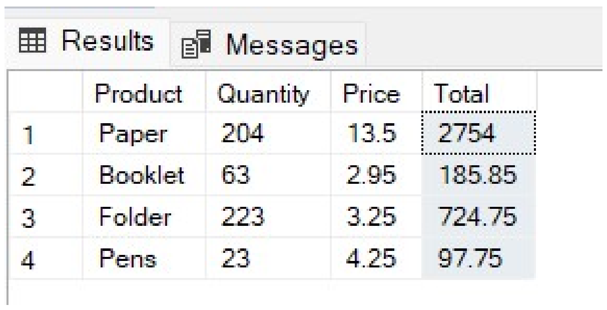


FIGURE 4.10: Product sales table hierarchy

After the data model has been created, a connection to the database can then be established to create the tables with their corresponding schema which is derived from the data model. The statement to set up the schema of the table is automatically generated and looks as shown below. The table is created with the according columns and their data types.

```sql
BEGIN
CREATE TABLE Product_Sales (
 [Product] NVARCHAR(255)
,[Quantity] FLOAT(53)
,[Price] FLOAT(53)
,[Total] FLOAT(53)
)
END
```

When a schema has been created, the data entries can be inserted. Thus, the table in the data model is written to the according table on the database. The result for the example can be seen in Figure 4.11. It can be seen that for each header, there is a column created and the data entries are correctly inserted row by row.



	Product	Quantity	Price	Total
1	Paper	204	13.5	2754
2	Booklet	63	2.95	185.85
3	Folder	223	3.25	724.75
4	Pens	23	4.25	97.75

FIGURE 4.11: Product sales table in database

5. Evaluating spreadsheet formulas

In the following Chapter, we outline the second stage of our headless spreadsheet approach, enabling us to evaluate formulas defined in the spreadsheet. The evaluation should not depend on the spreadsheet, but should be based on a representation of the spreadsheet that is built based on the tables in the database. Therefore, a way to build this internal representation of the spreadsheet within the software is required.

With this part of our headless spreadsheet approach, we aim to answer the second research question, which is about evaluating the formulas originally defined in the spreadsheet based on the database, i.e. without interacting with the spreadsheet document itself. For this purpose, it is necessary to find a suitable approach, which includes creating an internal representation of the spreadsheet at compile time and using it to evaluate formulas.

For formula evaluation, we can then apply the concept of headless spreadsheets. The internal spreadsheet representation is given as input to the instance of the headless spreadsheet. This allows us to make use of the formula evaluation engine of the headless spreadsheet.

5.1 Challenges

Building an internal representation of the spreadsheet to evaluate formulas introduces various challenges. Initially, it is to create an internal data structure representing the spreadsheet's cells. The internal representation needs to resemble the structure of the original spreadsheet and needs to be built according to it to be used for further processing.

Moreover, it is required to identify cells in the spreadsheet containing formulas, put them into context, parse them, and store them in a structured format. This format should be suitable for evaluation. Further, the formula evaluation engine that is used is required to process this format and calculate the results. This engine should support functions, operators, and references to other cells. The formula engine should be capable of processing the defined formulas and calculating the results accurately to handle formula evaluation. A comprehensive understanding of formula syntax, calculation precedence is crucial to ensure correct formula evaluation.

It is also to be noted that when the user changes the data within the generated application, the internal representation of the spreadsheet that is built needs to reflect the changes accordingly. It is required that the changes are reflected correctly for the formula evaluation to be correct on the new data. This requirement is important to guarantee the correct evaluation of the formula, as the data might be changed within the application frequently. Thus, it must be possible to make changes to the database and build a spreadsheet representation that reflects the changes. The changes can be inserts, updates or deletes of records.

Additionally, the formula evaluation process should be optimised for performance, as the goal of the headless spreadsheet approach is to achieve better performance than the spreadsheet approach. Therefore, techniques such as lazy evaluation, caching, and parallel processing must be considered for the formula evaluation engine. Lazy evaluation ensures that formulas are only recalculated when necessary, improving efficiency. Caching allows the application to store and reuse previously calculated results, reducing redundant computations. Parallel processing can take advantage to speed up calculations further.

5.2 Overview and architecture

We present an approach to build an internal representation of the spreadsheet and make use of it with the help of headless spreadsheets as a formula calculation engine. More specifically, we can use an instance of a headless spreadsheet, which enables us to evaluate spreadsheet formulas defined in the internal representation of the spreadsheet.

The process starts with a request to evaluate the formulas. Upon request, the internal representation is built based on the tables that have been previously transferred to the database. This representation is given as input to the headless spreadsheet instance, allowing for evaluating the formulas. The user can further change the data, which is reflected in the built spreadsheet representation. Thus, the user potentially gets different results from the formula evaluation according to the changes made.

As the spreadsheet representation is built according to the tables in the database, we have to access the tables and retrieve the data accordingly. Each record in the table was originally transferred from the database. Therefore, we can extract each record and consider them as data entries according to our data model introduced in 4.2. In this sense, we are reading the records from the tables and create a data entry object for each record, resulting in a list of data entries. Consequently, we load the records from the database to our data model. Thus, we proceed just the opposite way as we did when transferring the data from the spreadsheet to the database. There, we read the data entries from the spreadsheet to our data model and transferred it to the database. Now, at this stage, we are starting from the records from the database and load it into our data model. We can then build the internal spreadsheet representation which our headless spreadsheet requires. The architecture of the approach is given in Figure 5.1.

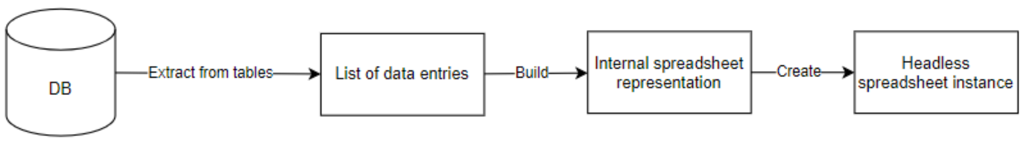
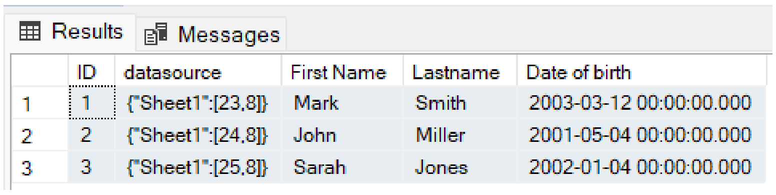

DB
Extract from tables
List of data entries
Build
Internal spreadsheet representation
Create
Headless spreadsheet instance

FIGURE 5.1: Evaluating spreadsheet formulas

5.3 Building internal spreadsheet representation

To build the internal representation of the spreadsheet based on the records in the database, it is required to have information available about where the respective record was located in the original spreadsheet. When retrieving the data from the database, we need to have this information prepared. For this purpose we add an additional column to each table in the database called "datasource". The column contains the sheet name and the row and column coordinates of where the data was stored in the original spreadsheet. An example of a data table containing the datasource coumn is shown in Figure 5.2. It can be seen that it contains the datasource column which stores the sheet name and the coordinates of where the record was originally defined in the spreadsheet. The coordinates are for the first record right to the datasource column. The coordinates for the other records within the same table row can be deducted by incrementing the column coordinate for each table column.



	ID	datasource	First Name	Lastname	Date of birth
1	1	{"Sheet1":[23,8]}	Mark	Smith	2003-03-12 00:00:00.000
2	2	{"Sheet1":[24,8]}	John	Miller	2001-05-04 00:00:00.000
3	3	{"Sheet1":[25,8]}	Sarah	Jones	2002-01-04 00:00:00.000

FIGURE 5.2: Data table including datasource column

The process of setting the datasource column can be performed during the transfer of the data from the original spreadsheet to the database. More specifically, during table detection and recognition as shown in Figure 4.7. When a data entry is recognised, we set the coordinates and the respective sheet name of the entry in the object. This guarantees that the data is accurately captured and as well as aligned with table detection and recognition. In Figure 5.3 the Entry class is shown that apart from the coordinates also contains the name of the sheet where it was defined.

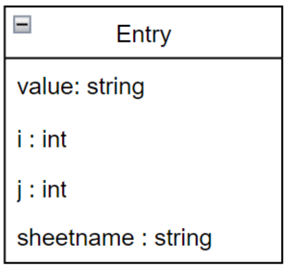


FIGURE 5.3: Entry class

Having the data tables prepared as described allows us to read this information from the database again when needed. When the formula evaluation is requested, for each data table, the records are loaded to the model as data entries, containing the coordinated and sheet name. From the data entry objects, we can start building the internal representation of the spreadsheet. The result needs to be a dictionary with a key-value pair, where the keys represent the names of the sheets and the values provide the content of the sheets in the form of a two-dimensional array.

During the process of building the internal representation, it is iterated over the data entries. Each data entry is placed in the two-dimensional array according to the set coordinates. The arrays are set as values to the keys defined as the according sheet names. Each data entry value is read from the database as a string, irrespective of its data type defined in the table. Formulas, too, are represented as expressions as strings without evaluation. The determination of data types is deferred to a later stage when the headless spreadsheet conducts content and format analysis. As a result, the entire sheet's cells are eventually represented in a comprehensive two-dimensional string array, preserving the dimensions of the original sheet.

5.4 Formula evaluation

After the internal representation of the spreadsheet has been built, it is ready to be used to create an instance of a headless spreadsheet. This instance serves as the engine that enables the formula evaluation. It allows to read specific cells within the internal representation and evaluate formulas, when it is given the respective cell address. After the user has uploaded a spreadsheet and intends to engage with its formulas, can thus initiate the formula evaluation on the headless spreadsheet instance, provided that the address of the formulas in interest is known.

In a spreadsheet, there can be many formulas, which might not all be interesting to be evaluated by the user. Therefore, it is important to provide users with the ability to specify which formulas should be available to be evaluated within the generated application.

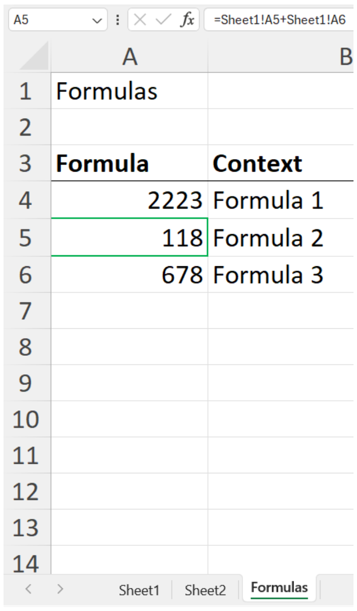
This level of control ensures that users can focus on the specific formulas relevant to their needs, thereby making the calculation process more efficient. It might be the case that only specific formulas within the spreadsheet are of interest, while others are not required to be available for evaluation within the web application. Thus, the user is provided with the possibility to specify which formulas are of interest. Moreover, valuable context information for the formulas can be provided by the user. This context information can take the form of formula names, descriptions, or any relevant metadata that aids in comprehending the purpose and function of each formula.

In our methodology, the formula evaluation process is facilitated by preparing the uploaded spreadsheet with the inclusion of an additional sheet named "Formulas". This specialised sheet allows to specify which formulas should be available in the generated application. The sheet contains the formulas that should be available for evaluation in the generated application and, offers a possibility for providing contextual information. Each row in the Formulas Sheet corresponds to a formula, specifying its expression or reference to the formula in the other sheets and its associated context information. This design creates a unified and organised structure that aligns with the user's intentions and simplifies the requirements engineering process.

In Figure 5.4, it can be seen how the Formulas Sheet is defined. It includes the formulas as well as context information as the formula name. The formulas take data from the other sheets as input.

The referenced formulas and their context information are systematically stored within a dedicated database table. The formulas are read and transferred to the database during table detection and recognition. This allows for efficient retrieval and evaluation of the formulas when required. It ensures that the formula evaluation process remains performance-optimised.

When the formula evaluation is triggered by the user, the selected formulas are evaluated. Therefore, on the headless spreadsheet instance, the Formulas Sheet is accessed and the cells containing the formulas are evaluated and the results provided.



A5 =Sheet1!A5+Sheet1!A6

	A	B
1	Formulas	
2		
3	**Formula**	**Context**
4	2223	Formula 1
5	118	Formula 2
6	678	Formula 3
7		
8		
9		
10		
11		
12		
13		
14		

Sheet1 Sheet2 Formulas

FIGURE 5.4: Formulas sheet

From a user perspective, the application can render the Formulas Sheet. Users can view the list of selected formulas, review the provided context information, and initiate formula evaluation as needed. This visual representation contributes to an efficient and user-friendly formula evaluation experience within the application.

5.5 Updating internal spreadsheet representation

As users interact with the generated application and manipulate the data within it, a fundamental challenge arises. This challenge is the need to build the internal representation of the spreadsheet according to the changes that are applied by the user. The generated representation needs to reflect the changes and correspond to the structure of the original spreadsheet. This includes the CRUD (Create, Read, Update, Delete) operations performed within the application. It is thus important to note that when making changes to the database tables, we need to be able to build a spreadsheet representation that is in accordance with the performed changes.

If we were not able to generate the internal spreadsheet representation in accordance to the performed changes, it would lead to any changes made to the data within the application not being reflected for the formula evaluation. The formula evaluation process is fundamentally reliant on the correctness of the internal spreadsheet representation. The consequence of a wrong spreadsheet representation would be a discrepancy between the data visible to users and the data being used by the formula engine, potentially leading to false calculations.

To address this challenge, a methodical approach is required. When changes to the data are made, we need to consider changing the datasource column as well. It stores the sheet name and the coordinates where the data needs to be located within the two-dimensional array.

In the case of an update operation on the data within the application, the respective records in the data table can be updated according to changes made. In this case, we do not have to update the datasource column as the sheet name and the coordinates remain the same. The updates made by the user are captured and preserved at the same location within the internal representation.

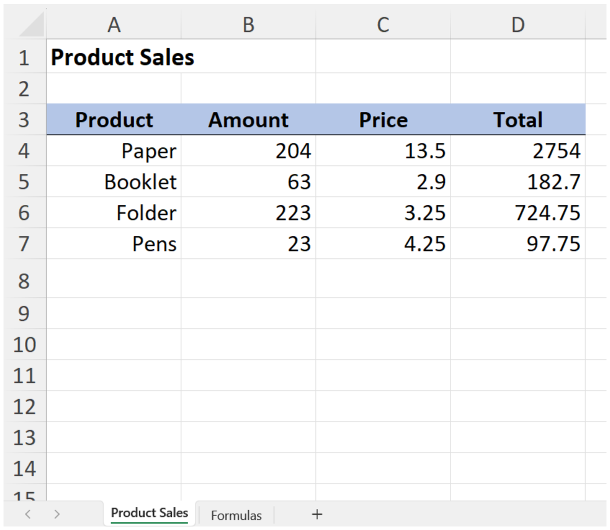
In cases of row deletions, the approach involves deleting the row in the database table. During the building of the internal spreadsheet representation, the deleted rows are not set in the two-dimensional array. They are either defined as null values or empty strings, indicating an empty cell within the spreadsheet. Including empty cells instead of entirely deleting the row ensures that the internal representation remains consistent and that the deleted data does not have an unintended impact on formula evaluation. Having empty strings ensures that the cell addresses of the rest of the sheet remain unchanged. Also, it is not required to change the datasource column of the subsequent rows.

In the case of inserting a new row into a data table, a new value for the datasource must be set. In this case, the datasource value needs to be set according to its appropriate location within the internal spreadsheet representation to be built. The insertion process needs to be performed carefully to maintain the structure and integrity of the internal representation. For this purpose, we take the coordinates of the last row and increment the row coordinate by one. This way, it is ensured that the new row is added after the last row of the table. Accordingly, the new entry also has the coordinates stored in the datasource column.

5.6 Example

In the following section, we illustrate the use of our formula evaluation approach through a small example. This example revolves around a prepared spreadsheet that is uploaded to the application to generate functionality from it. The tables included in the spreadsheet are transferred to the database. Further, it is possible to generate the internal spreadsheet representation from the data tables to build a headless spreadsheet instance for formula evaluation.

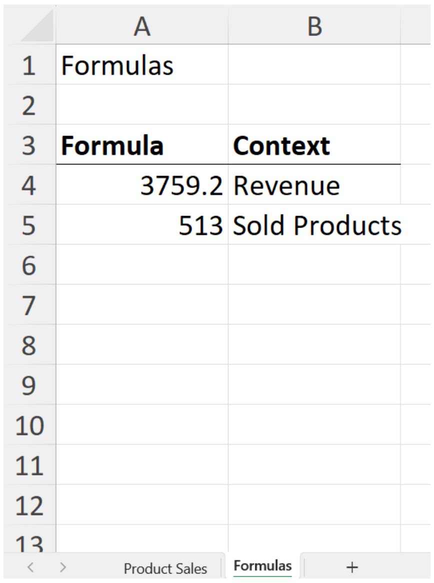
The spreadsheet used in the following example is shown in Figure 5.5. It has a sheet called "Product Sales" featuring a table. The table has a column "Total" which includes a formula for calculating the product of the "Amount" and "Price" column. This formula is not considered for evaluation in the generated application, as it is not specified in the Formulas Sheet. However, the resulting value is considered.



	A	B	C	D
1	Product Sales			
2				
3	Product	Amount	Price	Total
4	Paper	204	13.5	2754
5	Booklet	63	2.9	182.7
6	Folder	223	3.25	724.75
7	Pens	23	4.25	97.75
8				
9				
10				
11				
12				
13				
14				

FIGURE 5.5: Product Sales Table

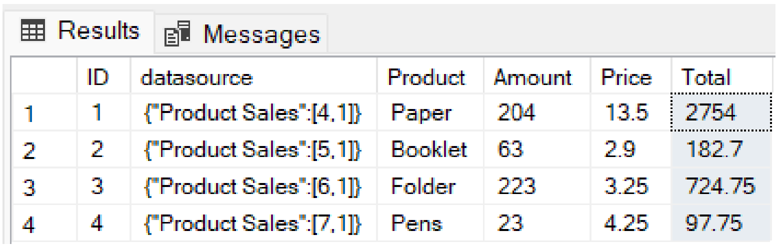
This spreadsheet, in addition to the product sales data, includes the dedicated Formulas Sheet. Within this sheet, formulas are defined to be evaluated within the application to be generated. The Formulas Sheet also contains the context of each formula, such as the formula name. In Figure 5.6 the content of the Formulas Sheet is given. It includes two formulas with the respective formula name.



	A	B
1	Formulas	
2		
3	**Formula**	**Context**
4	3759.2	Revenue
5	513	Sold Products
6		
7		
8		
9		
10		
11		
12		
13		

FIGURE 5.6: Formulas Sheet

After the user has successfully uploaded the spreadsheet, the next step involves setting up the tables within the database. The result can be seen on the example of the "Product Sales" data table in figure 5.7. The columns have been created according to the detected data types. For the column "Amount" not the formula is considered, but the calculated result. Formulas are only considered in the Formulas Sheet. Further, it can be noted that this newly created table not only contains the defined columns but also includes an additional column called datasource that is in place for each data table.

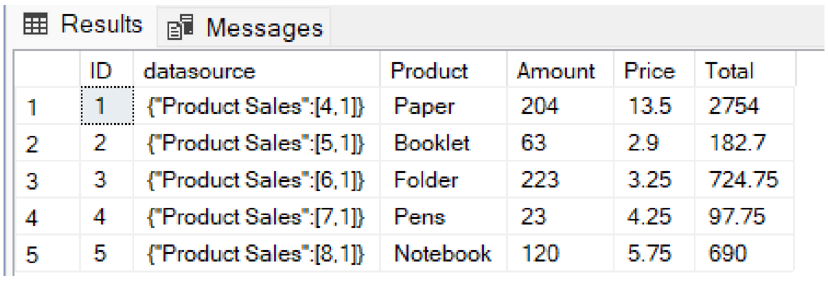


	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	2	{"Product Sales":[5,1]}	Booklet	63	2.9	182.7
3	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
4	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75

FIGURE 5.7: Product sales table on database

The datasource column plays an important role in this context. It stores the coordinates of the location of the corresponding entry within the original spreadsheet. In essence, it acts as a reference to the position of the data within the spreadsheet's internal representation. This additional information is crucial for building the internal representation required for formula evaluation. It ensures that every data record is accurately mapped to its original location within the spreadsheet.

When performing changes on the data within the generated web application, we must also consider the datasource column of the respective data table. For example, when adding a new row to the Product Sales table, we also have to add a value for the datasource. In Figure 5.8 an example is shown where we added a new row for the Notebooks to the Product Sales table. It can be seen that the new row has the same value for the data source as the previous rows, only differing in the row coordinate, which is increased by one compared to the row above it.



Results | Messages

	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	2	{"Product Sales":[5,1]}	Booklet	63	2.9	182.7
3	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
4	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75
5	5	{"Product Sales":[8,1]}	Notebook	120	5.75	690

FIGURE 5.8: Product sales table on database containing new row

After the spreadsheet has been transferred to the database, the formulas can be evaluated. For this purpose, the internal representation of the spreadsheet is built based on the data in the database. The internal representation is the foundation for formula evaluation.

It can further be seen, that the internal representation is set up as a dictionary. Each sheet in the spreadsheet corresponds to a key within this dictionary. Regardless of the data type, whether text, numbers, or formulas, the entirety of the sheet's data is uniformly stored in a two-dimensional string array. For each sheet, there is such a string array set as value in the dictionary. Within the two-dimensional string array representing the Formulas Sheet, formulas are represented in their original expression format, awaiting evaluation. Further, empty cells are defined as empty strings. In the code snipped below, for our example, the internal representation of the spreadsheet is given.

```
const sheets = {
 'Product Sales': [
   ['Product Sales', '', '', ''],
   ['', '', '', ''],
   ['Product', 'Quantity', 'Price', 'Total'],
   ['Paper', '204', '13.5', '2754'],
   ['Booklet', '63', '2.95', '182.7'],
   ['Folder', '223', '3.25', '724.75'],
   ['Pens', '23', '4.25', '97.75'],
   ['Notebook', '120', '5.75', '690'],
  ],
 'Formulas': [
   ['Formulas, ''],
   ['', ''],
   ['Formula', 'Name'],
   ['=SUMME(Tabelle2!B5:B8)', 'Sold Products'],
   ['=SUMME(Tabelle2!D5:D8)', '3', 'Revenue'],
  ],
};
```

This internal representation is the foundation for building the headless spreadsheet instance, enabling formula evaluation. The Formulas sheet captured within this representation enables the application to perform dedicated formula calculations defined by the user. When the evaluation is triggered, an instance of a headless spreadsheet is built based on the database tables and the respective Formulas Sheet is accessed within the headless spreadsheet instance, allowing to evaluate the formula expressions defined therein.

Importantly, should changes to the data occur, the internal representation automatically reflects these changes. This ensures that the data used for formula evaluation remains up-to-date, preserving the integrity and accuracy of the calculations throughout the application's lifetime.

6. Web application showcase

In the following Chapter, we introduce a web application that makes use of the headless spreadsheet approach. This allows us to see how it could work out in practice. Further, we outline its architecture.

6.1 Goal

With our web application, we want to provide a practical approach to make use of the many spreadsheets, currently unused in the software development process. We want to provide a possibility that spreadsheets that have been prepared as outlined in section 5.4 can be used for requirements specification for automatic software generation. Within the web application, we provide the possibility to upload a spreadsheet. The new functionality is generated based on the uploaded spreadsheet. This includes transferring the data from the spreadsheet to the attached database and evaluating the formulas defined in the spreadsheet based on the data in the database. It is further possible, to make changes to the data in the database and evaluate the formulas according to the changes.

6.2 Web application introduction

In the following section, the web application is introduced. Starting from the process of initial spreadsheet upload to the execution of formula evaluation and performing changes on the data. Parts from the application's UI are presented to provide a comprehensive understanding of how the methodology can work in a practical scenario.

The web application allows to uploading a spreadsheet based on which the functionalities of the web application are generated. In Figure 6.1, it can be seen how the home screen of the web application looks like. It allows to upload a spreadsheet. For our methodology, it is required to have the formulas which should be made usable within the web application prepared in the dedicated Formulas sheet as outlined earlier in Figure 5.4.

In Figure 6.1, it can further be seen that the Table View and the Formula View can be accessed in the navigation menu. Initially, the Views are empty. As soon as a spreadsheet is uploaded, the Views are populated with the new functionalities.

After the initial upload of the spreadsheet, the tables are detected and recognised and the formulas from the dedicated sheet are read. All the data is imported to the attached database accordingly. Users can then interact with the imported data, making modifications to it that potentially impact the formula calculations. This allows to work with the data and formulas without modifying the original spreadsheet.

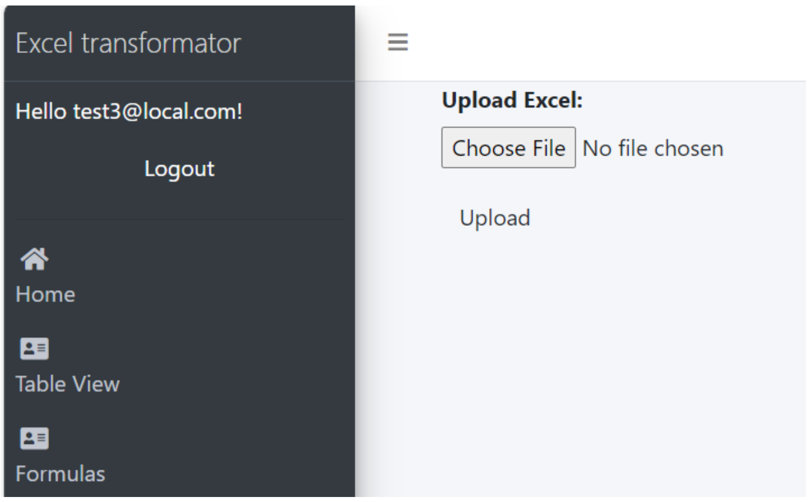


FIGURE 6.1: Home screen

The user can get an overview of the detected and recognized tables within the Table View of the web application. In the example shown in figure 6.2, four tables were detected in the spreadsheet. There is the possibility to delete or modify tables.

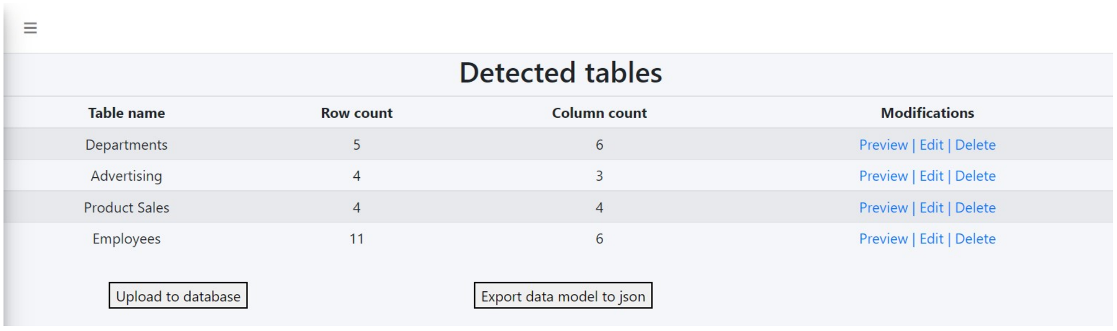
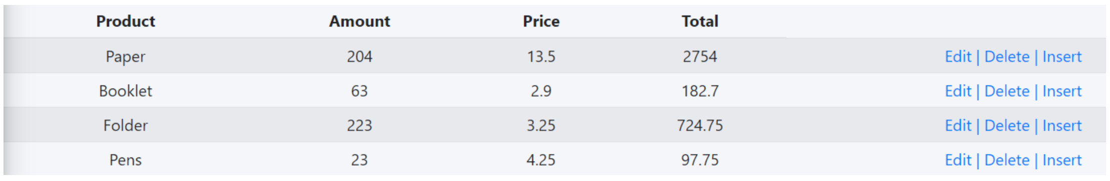

Detected tables

Table name	Row count	Column count	Modifications
Departments	5	6	Preview | Edit | Delete
Advertising	4	3	Preview | Edit | Delete
Product Sales	4	4	Preview | Edit | Delete
Employees	11	6	Preview | Edit | Delete

FIGURE 6.2: Table view with detected tables listed

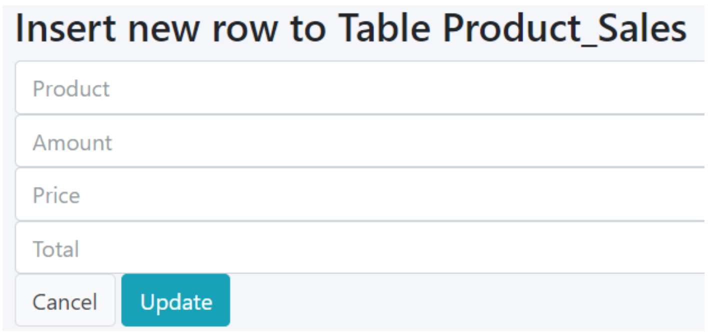
Further, there is the possibility of getting a Preview of each detected table that includes the columns and data entries. It allows the user to check the table and make changes to it. There is the possibility to delete, update or insert new rows. An example of a preview of a table is given in figure 6.3. This View allows us to make changes to the data. Changes made are later considered during formula evaluation.



Product	Amount	Price	Total	
Paper	204	13.5	2754	Edit | Delete | Insert
Booklet	63	2.9	182.7	Edit | Delete | Insert
Folder	223	3.25	724.75	Edit | Delete | Insert
Pens	23	4.25	97.75	Edit | Delete | Insert

FIGURE 6.3: Preview of a particular table

When the user wants to create or update data, a form is opened that allows them to input new data. Figure 6.4 illustrates how the form to insert a new row to a table looks like. For each column of the table, a value can be entered that must match the underlying data type. Thus, the input of invalid data is prohibited. After using the Update button, a new row is created for the respective data table.

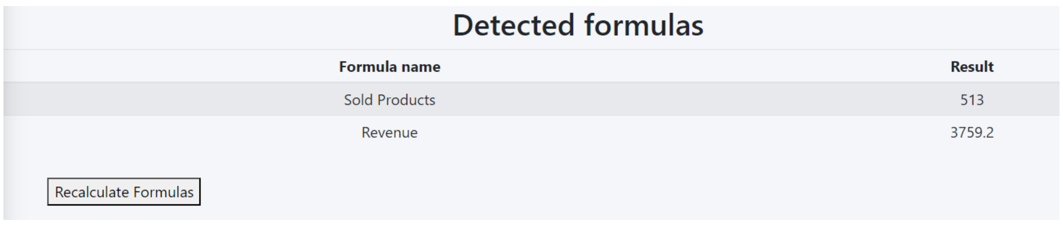

Insert new row to Table Product_Sales

Product
Amount
Price
Total

FIGURE 6.4: Form to insert new data row

In Figure 6.5, it can be seen what the formulas View looks like. The formulas that have been defined in the spreadsheet are given there. Further, there is a button to recalculate formulas. If changes to the data have been made, this button needs to be triggered to obtain the results. Upon button call, the internal spreadsheet representation is created based on the database. Further, the headless spreadsheet instance is built and the results from the formula calculations are read and stored in the database.

It needs to be noted, that if the data on which the formulas depend change and the formula evaluation is not triggered, the results shown are not up to date. Thus, when wanting to ensure to obtain the newest results, it is required to recalculate the results. Depending on the file size, this operation can be expensive. By providing the user with the possibility to decide when this calculation should be performed, flexibility is increased compared to implementations where the evaluation is automatically performed after every data change.


Detected formulas

Formula name	Result
Sold Products	513
Revenue	3759.2

FIGURE 6.5: Formulas View

6.3 Architecture

The headless spreadsheet approach is implemented in a web application such that it can be used by any logged-in user to upload a spreadsheet document which is to be transferred to a database to work more efficiently with the data. To achieve a clear separation between different components, the MVC design pattern [30] is used for the web application. Following this pattern, the spreadsheet document serves as Model from which a data model is generated as the internal representation of the Model. The Controller handles user requests and updates the Model if needed. Also, the Controller passes the model to the Views to render it to the user. Figure 6.6 shows how the architecture looks like.

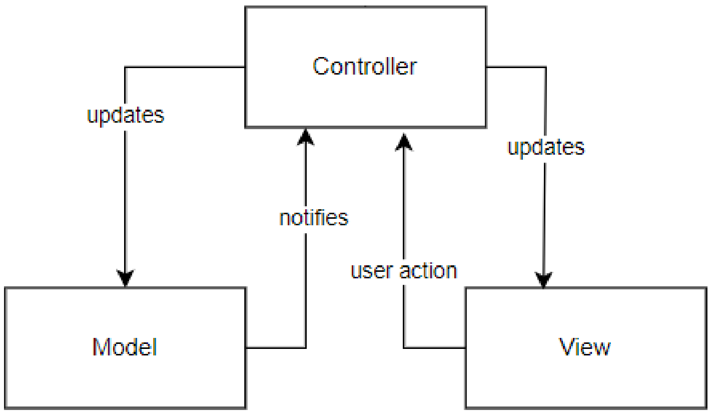


FIGURE 6.6: MVC

6.4 Component diagram

To show the structural organisation of the web application in more detail, in Figure 6.7 we outlined the components that make up the system and their interactions in more detail. It provides more insight into the components of the MVC pattern.

Also, it can be that there are services which provide different functionalities. The Spreadsheet Service is used during the initial upload of the spreadsheet to create the data model. The Database Service then provides functionalities to transfer the Model to the database and make changes to it. Further, there is a Headless Spreadsheet Service which includes the functionality for formula calculation such as building an internal spreadsheet representation for the headless spreadsheet instance required for formula evaluation.

In the View component, the different Views as introduced can be seen. The Views allow the user to request the creation of the data model or to update it. Also, from the Formula View, the user can request the formula calculation.

The requests from the users are handled by the three Controllers which are the Home, Table and Formula Controller. The Home controller is requested during the initial upload of the spreadsheet. It makes use of the Spreadsheet and the Database service to transfer the tables from the spreadsheet to the database. Further, the Table controller is in charge of model updates. It uses the Database Service to make changes to the database when requested by the user. The formula Controller on the other hand handles formula evaluation. For this purpose, it makes use of the Headless Spreadsheet Service which provides the internal spreadsheet representation for formula evaluation.

On the Model component, it can be seen that the Model consists of tables, titles, column headers and data entries. These classes are required to represent the tables of the original spreadsheet. As outlined in Chapter 4 we have a hierarchical clustering of the classes. Each table object represents a table from the spreadsheet and consists of a title, columns and data entries. A table object composes a list of columns, while each column object composes a list of data entries. The data entries that are assigned to a column represent the data that forms a column within the original spreadsheet.

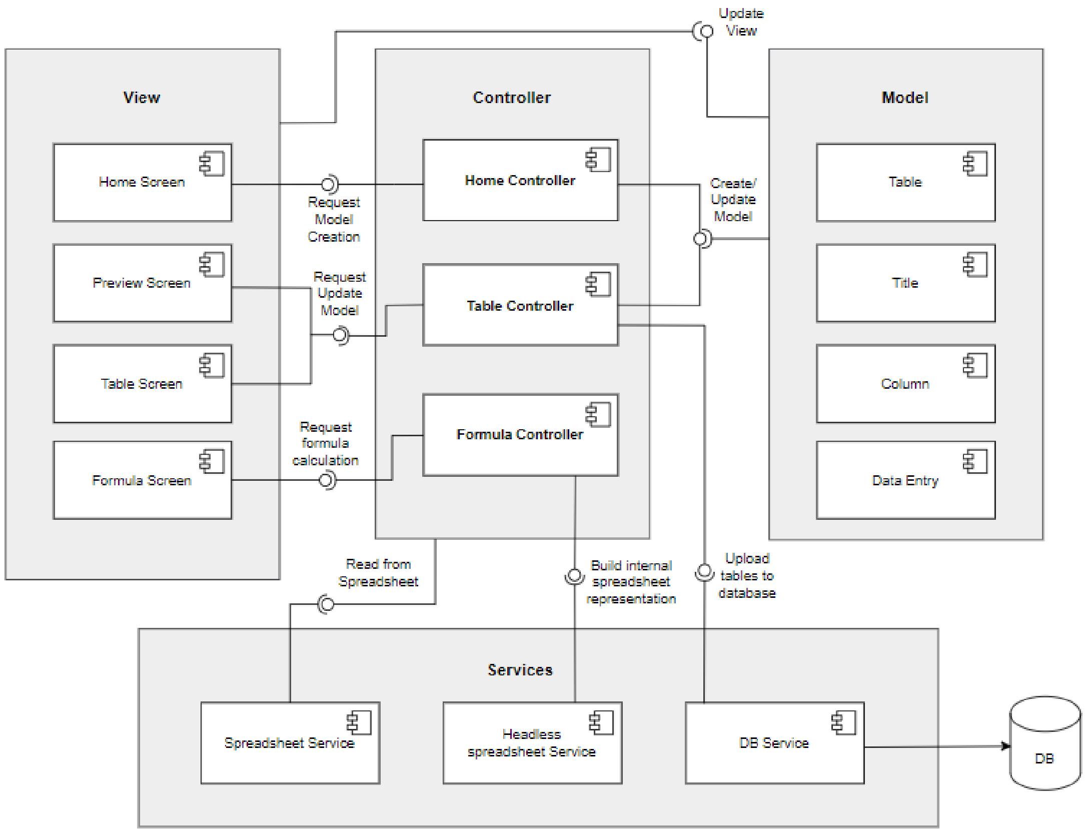


FIGURE 6.7: Component diagram

6.5 Sequence diagrams

To further illustrate how components interact over time, we provide sequence diagrams, helping to understand the flow of control and the order of actions. They illustrate how components collaborate to fulfil specific use cases. The use cases for which we present the sequence diagram are the upload of a spreadsheet, updating the data model and formula calculation.

The first sequence diagram is illustrated in Figure 6.8. It shows the flow of actions involved when uploading the initial spreadsheet. When the spreadsheet is uploaded on the Home View, the Home Controller calls the Spreadsheet Service to create the Model. Further, the Home Controller makes use of the Database Service in order to store the newly created model in the database. After this process has finished, the tables from the spreadsheet have been transferred to the database.

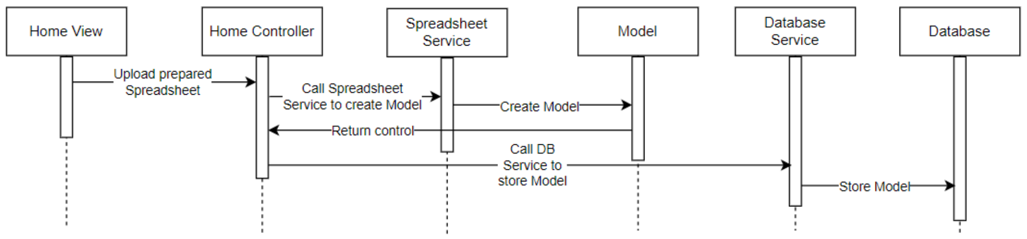


FIGURE 6.8: Upload spreadsheet

The next sequence diagram illustrated in Figure 6.9, shows the actions involved when making changes to the data model. The process starts in the Preview or in the Table View. Users can there request changes of the data model. When a change of the data model is requested, the Table Controller is in charge of making the changes requested by the user to the database. For this purpose, the Table Controller makes use of the Database Service which provides the functionalities to make changes to the database. After the changes have been performed, the new data model can be extracted from the database and be provided to the View such that the user can see the changes.

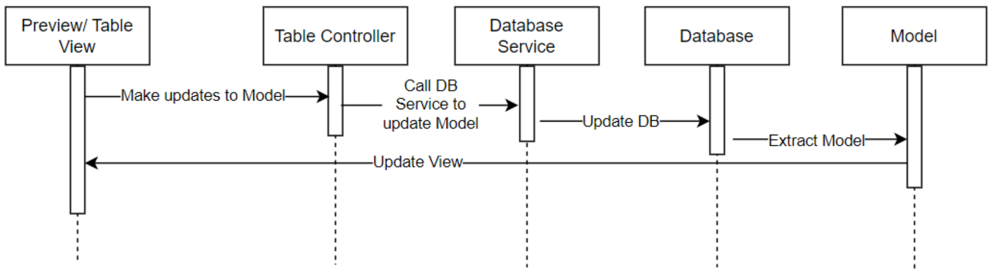


FIGURE 6.9: Make changes to data model

Finally, in the sequence diagram shown in Figure 6.10, the actions involved when requesting a recalculation of the formulas is provided. The process starts on the Formula View, where the user can request the formula evaluation via button. When the button is triggered, the request is handled by the Formula Controller which calls the Headless spreadsheet Service. The service requests the required data tables from the database to build an internal representation of the spreadsheet and returns it to the Formula Controller. The Formula Controller can then build the headless spreadsheet instance and evaluate the formulas. The results are finally provided to the Formula View, where the results can be seen.

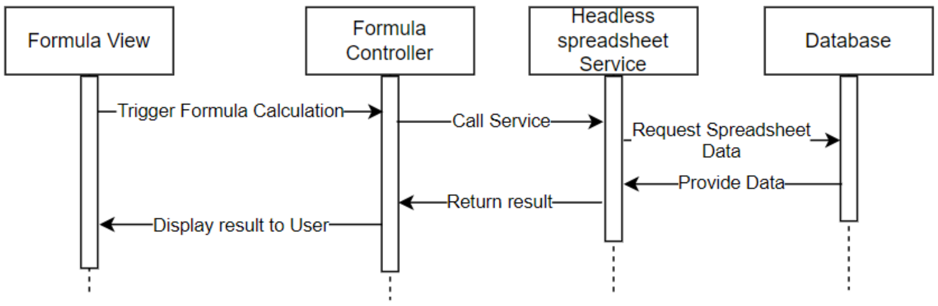


FIGURE 6.10: Request formula evaluation

6.6 Learning outcomes

The showcase outlines how the headless spreadsheet approach can be applied to use spreadsheet documents as a requirements specification for automatic software generation. Users are provided with the possibility to generate functionality based on a prepared spreadsheet that is uploaded to a web application. Subsequently, they can work on the data from the spreadsheet within the web application.

Starting with uploading the prepared spreadsheet to the web application, it is outlined how table detection and recognition are applied to the spreadsheet to transfer the tables within the spreadsheet to the attached database. The creation of a data model from the uploaded spreadsheet and its subsequent transfer to an attached database is a key aspect of the headless spreadsheet approach.

After the data has been transferred to the database, users can evaluate the formulas originally defined in the spreadsheet in a dedicated formulas sheet. The evaluation is performed based on the database. Thus, the original spreadsheet is not required anymore. For this purpose, the internal spreadsheet representation is built. It allows to create an instance of a headless spreadsheet that can be used for formula evaluation.

As soon as the spreadsheet is uploaded, the data can be modified within the application as necessary. The internal spreadsheet representation is built according to the changes when re-evaluating the formulas. Thus, the formula evaluation is performed according to the new data.

Finally, the web application showcase outlines how the headless spreadsheet approach could be used in practice. As a demonstration of the approach's usage, the showcase serves as a bridge between theory and application. It serves as a foundation for further refinement and adaptation based on real-world use cases.

7. Implementation details

In the following Chapter, we outline the main parts of our proposed implementation for the web application using the headless spreadsheet approach as demonstrated in Chapter 6. The implementation details are provided according to the two stages of the headless spreadsheet approach which are transferring spreadsheet tables to a relational database and formula evaluation.

The implementation is in C# .Net. Further, Epplus is used to read the original spreadsheet during the import. As Epplus only allows it to be used with Excel, the focus of our implementation is on Excel documents. All the implementation details are provided on a GitHub repository [19].

7.1 Transferring Excel tables to a relational database

In the following section, we outline the main parts of the implementation of the first stage of our methodology, which includes our implementation of the described rule-based algorithm for table detection and recognition, as well as the transfer of the data model into a database and further challenges. We also outline how we solved the challenge of handling different data types and missing values.

7.1.1 Table detection and recognition

In our proposed implementation, we have provided the logic for table detection and recognition in the dedicated spreadsheet service. The service is instantiated with the path to the Excel document. After uploading a spreadsheet, an instance of the ExcelPackage of EPPlus is created to read the document. When instantiating the service, the algorithm for table detection and recognition is automatically performed and the tables are stored in a field of the class. The final data model is also available within the object.

The method provided below performs the part of table detection as described. For each sheet within the Excel document, it gets the dimension of the sheet (lines 11 and 13) which serves as a starting point for the algorithm. Subsequently, in line 25, the table recognition is performed on each cell of the sheet individually, iterating from the bottommost, rightmost cell to cell C[0,0]. It then returns the created tables.

```
1 public List<Table> TableDetection()
2 {
3     foreach (var ws in wss)
4     {
5         int colCount;
6         int rowCount;
7 
8         try
9          {
10             //get column count of document
11             colCount = ws.Dimension.End.Column;
12             //get row count of document
13             rowCount = ws.Dimension.End.Row;
14         }
15         catch
16         {
17             colCount = 0;
18             rowCount = 0;
19         }
20
21         for (int row = rowCount; row >= 1; row--)
22         {
23             for (int col = colCount; col >= 1; col--)
24             {
25                 TableRecognition(row, col, ws);
26             }
27         }
28     }
29     return tables;
30 }
```

For the table recognition part, the corresponding method is called with the coordinates of the current cell as well as with the current worksheet. If the current cell is null, i.e., an empty cell, the method returns without processing any further logic. Otherwise, it checks for the different cell types and creates an instance of the respective cell depending on which type it belongs. The cells can be assigned to either the data entries, columns or titles. The checks for the different cell types are performed in dedicated functions, which contain the logic according to the rules for table recognition as outlined. The checks return true or false depending on whether the current cell is of the respective type. The order of the checks is such that data cells are checked first, then header cells and finally title cells. In this order, table recognition is performed as efficiently as possible, since most cells are data cells, while title cells appear least frequently. In addition, each instantiated cell has a property "tableCounter", which is used to identify the assignment of a cell to a table. Subsequently, the columns can be assigned to the tables, while the data entries can be assigned to the columns based on their coordinates. The according assignments are created in lines 25 and 26 as well as lines 37 and 38.

```
1  public void TableRecognition(int i, int j, ExcelWorksheet ews)
2  {
3      if (ews.Cells[i, j].Value == null) return;
4
5      // Check for data cell
6      else if (IsDataCell(i, j, ews))
7      {
8          //Add new data cell
9          var newDataCell = new Entry();
10         newDataCell.tableCount = tableCounter;
11         newDataCell.i = i;
12         newDataCell.j = j;
13         newDataCell.value = ews.Cells[i, j].Value;
14         dataEntries.Add(newDataCell);
15     }
16
17     // Check for header cell
18     else if (IsHeaderCell(i, j, ews))
19     {
20         // Add new header
```

```
        var newColumn = new Column();
        newColumn.tableCount = tableCounter;
        newColumn.count = j;
        newColumn.Name = ews.Cells[i, j].Value;
        newColumn.entries = dataEntries
        .Where(e => e.j==j && e.tableCount==tableCounter).ToList();
        columns.Add(newColumn);
    }

    // Check for title cell
    else if (IsTitleCell(i, j, ews))
    {
        // Add new title
        var newTitle = new TableName();
        newTitle.tableCount = tableCounter;
        newTitle.name = ews.Cells[i, j].Value;
        newTitle.columns = columns
        .Where(C=> C.tableCount == tableCounter).ToList();
        tables.Add(newTitle);
    }
}
```

7.1.2 Transferring the data model to a database

After the data model is obtained, the tables need to be set up in a relational database accordingly. Microsoft SQL Server is used as a Database management System. There are other alternatives that could have been chosen as well. However, to stay within the .Net framework, SQL Server is chosen. In our proposed implementation, the logic for the database operations is performed by the Database Service.

Before tables can be transferred, a database must first be created in SQL Server. The Database Service contains the corresponding function to do so. It will create a new database called according to a name which can be set. If a corresponding database already exists, no new one is created. The corresponding SQL statement where the database name is given as a parameter is shown below. Alternatively, the database can also be created manually in the SQL Server Management Studio.

```
        // 0 = DB name
        private const string createDbCmd = @"
        IF NOT EXISTS(SELECT * FROM sys.databases WHERE name = '{0}')
        BEGIN
        CREATE DATABASE[{0}]
        END";
```

The SQL statement can be executed on the SQL Server with the help of Transact SQL statements [32]. They allow to execute a prepared SQL statement with the according parameters against the SQL Server. Thus, a connection to SQL Server has to be established. For this purpose, a session with the SQL Server has to be opened. The session corresponds to a network connection to the SQL Server. Therefore, a connection string [29] is required. The connection string contains information required to connect to the SQL Server. An example of a connection string is given below:

"Server=.;initial catalog="Test";Integrated Security=true"

There are server, initial catalog and integrated security information specified. The server is defined as "." which is equivalent to local. The initial catalog represents the name of the database to which a connection should be established. Moreover, Integrated security can be defined. In the example, it is set to true which means that Windows authentication is used to authenticate to the SQL Server.

With the help of the connection string, the Transact SQL statement can be executed on the SQL server. Below it can be seen how a connection [28] to the SQL Server is opened and a query executed. The command is prepared with the SqlCommand Class [31] as shown in line 6. The SQL query can be executed on this class as shown on line 8.

```
1  // Helper function to execute Sql commands
2  private static void ExecCommand(string queryString, string
       connectionString)
3  {
4      using (SqlConnection connection = new SqlConnection(
       connectionString))
5      {
6          SqlCommand command = new SqlCommand(queryString, connection);
7          command.Connection.Open();
8          command.ExecuteNonQuery();
9      }
10 }
```

As soon as a connection to a database has been established and the database has been created with the Transact SQL statement, the tables can be inserted into the database. For every table which is to be inserted, the corresponding schema has to be defined [23]. The required Transact SQL statements to create the schema are generated from the data model and stored in a string. In the below code example, the corresponding function is shown. A string builder is used to create the command. Each column of the tables is specified by appending a prefix to separate the columns, the column name and the datatype of the column (see line 16). This way the generic generation of the database schema is guaranteed. The command can be executed with an open connection to the SQL Server.

It must further be noted that if a table with the specified name already exists in the database, no new table is created. Instead, the data entries are inserted into the existing table. It is the user's responsibility to avoid duplicates.

```
1  public void CreateTable()
2  {
3      StringBuilder sb = new StringBuilder();
4      sb.AppendLine($"IF NOT EXISTS
5      (SELECT * FROM sysobjects
6      WHERE name='{tableName}' and xtype='U')");
7      sb.AppendLine("BEGIN");
8      sb.AppendLine($"CREATE TABLE {tableName} (");
9
10     for (int colIdx = 0; colIdx < table.columnCount; colIdx++)
11     {
12         string type = CheckDataType(1, colIdx);
13         string prefix = " ";
14         if (colIdx != 0) prefix = ",";
15         sb.AppendLine($"{prefix}[{table.columns[colIdx]}] {type}");
16     }
17     sb.AppendLine(")");
18     sb.AppendLine("END");
19 }
```

Afterwards, the data entries are inserted into the table with the help of bulk inserts [22]. For this purpose, a data table can be filled with data from the data model. Again, a connection to the SQL Server is required. Subsequently, the data table can be copied to the defined destination table on the SQL Server. Bulk inserts allow it to insert a large number of rows into a database table in a single operation. Instead of executing individual insert statements for each row, bulk inserts allow to efficiently insert multiple rows at once, which can greatly improve performance with large amounts of data.

The following code snippet shows how the insertion of a table is carried out. In line 6, the connection to the database is established, while in line 7 a new SQL transaction, i.e. an operation executed on the database, is started. With the help of a SqlBulkCopy object, which is instantiated in line 9, the data table "tbl" filled from the data model can then be written to the database.

```
1  public void TableInsert()
2  {
3      string connection = GetConnectionString("Test");
4      using (SqlConnection con = new SqlConnection(connection))
5      {
6          con.Open();
7          using (SqlTransaction transaction = con.BeginTransaction())
8          {
9              SqlBulkCopy objbulk = new SqlBulkCopy(con, transaction);
10             objbulk.DestinationTableName = tableName;
11
12             try
13             {
14                 objbulk.WriteToServer(tbl);
15                 transaction.Commit();
16             }
17             catch
18             {
19                 transaction.Rollback();
20                 throw;
21             }
22         }
23     }
24 }
```

7.1.3 Handling different data types and missing values

A further challenge in the process of transferring data from Excel documents to a relational database is handling different data types as well as missing values. Since data entries of tables in Excel documents can have different data types and missing values, the task is to identify them correctly and set up the database schema according to the data types. Usually, the data entries within a given column are of the same data type.

EPPLus allows it to read each data cell as type "object", which is the base class of all objects in C# [21]. To handle different data types, the data entries are stored as two-dimensional arrays of this object type. It allows that values of any type can be assigned. Thus, data entries are read from the Excel documents with the object type and assigned to the two-dimensional array of type object in the data model. The advantage of this approach is that the data type does not need to be specified when reading from the Excel document, but can be resolved later. This way it is ensured the data entries can be of any type within a specific column.

The second part of the challenge is to create the table schemas according to the type of the objects in the data model. The data model holds the objects which then need to be translated to a string that can be inserted into the command for the creation of the table schema. Therefore, there is a dedicated function required, which checks the type of an object and returns the according type which is supported by SQL Server [24] to be inserted in the command.

The function that performs this mapping is given below. It covers the most common data types in SQL Server. It is called once for each column of the table. At first, in line 3, the type of the data entries is extracted which can then be matched to a string that matches this type best (lines 5-10). Data types that can not be matched will be in any case identified as NVARCHAR(255).

```
1  public string CheckDataType(int i, int j)
2  {
3      Type type = table.values[i, j].GetType();
4  
5      if (type == typeof(int)) return "INT";
6      else if (type == typeof(double)) return "FLOAT(53)";
7      else if (type == typeof(float)) return "FLOAT(53)";
8      else if (type == typeof(decimal)) return "DECIMAL(53, 2)";
9      else if (type == typeof(DateTime)) return "DATETIME";
10     else return "NVARCHAR(255)";
11 }
```

In practice, it is often the case that tables in Excel documents are incomplete, i.e., some data entries are missing. The rule-based approach presented by Doush and Pontelli [7] does not contain information about handling missing values. However, for our proposed implementation of their rule-based approach, missing values could possibly cause incorrect recognition of the table. Thus, it is required to give a special emphasis to missing values in tables and adjust the Spreadsheet Service to be able to handle them.

For our methodology, we decided to treat missing values as described in the following. The user is required to replace empty cells in tables of the Excel document with either "n/a" or "N/A". We have extended the Spreadsheet Service, such that if one of the values is identified, the according cell is handled as data entry with the value null. The respective null value is eventually inserted into the database accordingly. However, it is the user's responsibility to correctly fill in missing values. If the user does not, some unwanted errors in table recognition might appear.

7.2 Formula evaluation

In the following section, we outline the implementation of the second stage of the headless spreadsheet approach, which allows formula evaluation. For this purpose, we need to prepare some information. This information is required to build the internal spreadsheet representation. We further focus on the process of building of the internal spreadsheet representation and how it is used for formula evaluation with the help of a headless spreadsheet instance.

7.2.1 Preparation steps

To build the internal spreadsheet representation, some information is required and needs to be prepared. On the one hand, we need to have the defined formula expressions in the original spreadsheet on the Formula Sheet available in a dedicated table. This is important to be able to extract the formulas and not the evaluated results. On the other hand, we need to know which data tables need to be included in the internal spreadsheet representation and where the respective records need to be placed.

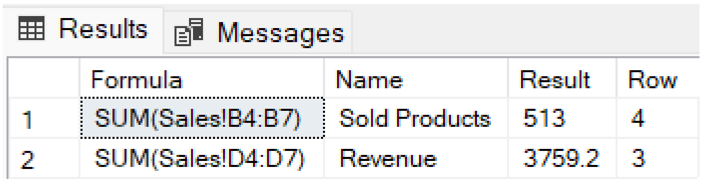
Formulas table

To have the formula expressions defined in the original spreadsheet readily available, we need to store them in a dedicated table. We can not treat the formulas as a regular table, as we want to extract the formula expressions from the cells and not the value, i.e. the evaluated result. This further gives us the possibility to store meta-data within this table and calculation results, which can be later retrieved such that it does not need to be recalculated.

For this purpose, it is required to create a table storing the relevant information during the initial import of the spreadsheet. Thus, when we iterate over the sheets of the imported spreadsheet, we can detect the Formulas Sheet and retrieve the required information from it. The according code snippet is given below and shows the functionality defined in the Spreadsheet Service. It can be seen that if the Formulas Sheet is identified, the relevant information is retrieved and stored as formula object. A list of the formula object is then used as shown in line 16 to store the formulas on a dedicated formula table. Further, in line 11, we can read the formula expression from the cell with the help of the "Formula" property. This ensures that we read the formula expression and not the evaluated result.

```
1  foreach (var ws in wss)
2      {
3          if (ws.Name == "Formulas")
4          {
5              List<Formula> formulas = new List<Formula>();
6              for (int row = rowCount; row >= 4; row--)
7              {
8                  if (ws.Cells[row, 1].Formula == ""
9                  && ws.Cells[row, 1].Value == null) continue;
10                 var newFormula = new Formula();
11                 newFormula.formula = ws.Cells[row, 1].Formula ?? null;
12                 newFormula.result = ws.Cells[row, 1].Value?.ToString();
13                 newFormula.name = ws.Cells[row, 2].Value?.ToString();
14                 newFormula.i = row-1;
15                 formulas.Add(newFormula);
16             }
17             dbService.CreateFormulaTable(formulas);
18         }
19     }
```

In Figure 7.1, an example of the according Formulas table is given. It includes the formula expression as well as the evaluated result and additional context information. Preparing this information in a dedicated table, allows us to later retrieve the calculated results if needed. Also, it allows us to extract the formula expression for building the internal spreadsheet representation.

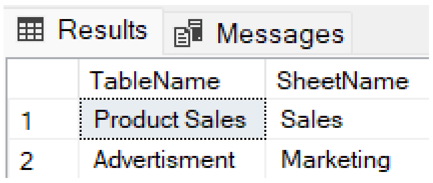


	Formula	Name	Result	Row
1	SUM(Sales!B4:B7)	Sold Products	513	4
2	SUM(Sales!D4:D7)	Revenue	3759.2	3

FIGURE 7.1: Formulas table

Preparing metadata and datasource

To know which tables need to be included in the internal spreadsheet representation, we provide this information in a dedicated table called "Metadata". It includes information about which tables have been detected in the original spreadsheet. Also, the sheet name where the table was detected in the original spreadsheet is given. We can later use this information to query the data tables which are given in the Metadata table. An example of the table is given in Figure 7.2. There are two rows given, indicating that in the uploaded spreadsheet, there have been two tables detected.



	TableName	SheetName
1	Product Sales	Sales
2	Advertisment	Marketing

FIGURE 7.2: Metadata table

The function to create the Metadata table is given in the database service. It needs the detected tables as input. Thus, we can use this function after we have successfully detected and recognised the tables. The function to create the Metadata table is given below. It shows a prepared statement that creates a new table Metadata if it not already exists and adds the metadata to the table.

```
public void CreateMetadataTable(List<TableName> tables)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine($"IF NOT EXISTS (SELECT * FROM sysobjects
    WHERE name='Metadata' and xtype='U')");
    sb.AppendLine("BEGIN");
    sb.AppendLine($"CREATE TABLE Metadata (");
    sb.AppendLine($"[TableName]  NVARCHAR(MAX),");
    sb.AppendLine($"[SheetName]  NVARCHAR(MAX)");
    sb.AppendLine(")");
    sb.AppendLine("END");
    ExecCommand(sb.ToString(), GetConnectionString("Test"));
    InsertMetadata(tables);
}
```

We further have to include the datasource column in each detected table to be able to know where the data needs to be placed within the internal spreadsheet representation we want to build. We can add this operation in the table detection and recognition algorithm defined in the Spreadsheet Service. When we found a data entry in the spreadsheet, we add the operations as shown below. We create a new dictionary for the datascource, where we add the sheet name as key and the coordinates of the data entry within that sheet as value. We can then add this information to the data entry object. This allows us to include the datasource to the tables in the database where we can retrieve it again when we need to build the internal spreadsheet representation.

```
var data = new Entry();
Dictionary<string, int[]> datasourceInfo =
new Dictionary<string, int[]>();
int[] datasourceValue = new int[2] { i, j };
datasourceInfo.Add(ews.Name, datasourceValue);
data.datasource = datasourceInfo;
```

7.2.2 Building internal spreadsheet representation

After we have prepared the Metadata table and set the datasource in the data tables, it allows us to build the internal spreadsheet representation based on the data tables.

In the code snippet below we outline the Process method from the Headless Spreadsheet Service. The method is responsible for the build process of the internal spreadsheet representation. For this purpose, it iterates over the detected tables given in the Metadata table. For each table, it uses the table reader as shown in line 15 to read all records from the table and store the values as well as the datasource information back to a data entry object which was also used during the import of the original spreadsheet to transfer the data to the database. For each record within the data table, we create a data entry object and read the data from the table as shown from line 24 to 38. All the data entry objects are then added to a list.

After we have obtained the list of data records form the table and their according datasource information, we are able to build the internal spreadsheet representation based on this list. The required method for this task is the BuildInternalRepresentation method which is called within the Process method as shown in line 43.

```
1  public Dictionary<string, string[,]> Process()
2  {
3      var data = new List<DataEntry>();
4
5      // Iterate through tables in metadata
6       foreach (var table in _metadata.Keys)
7       {
8          // SQL query to retrieve data from each table
9          string query = $"SELECT datasource, * FROM {table}";
10         if (table == null) continue;
11
12         new SqlCommand(query, connection))
13         using (SqlDataReader reader = command.ExecuteReader())
14         {
15             while (reader.Read())
16             {
17                 var sheetName = _metadata[table];
18                 var sourceValue = reader["datasource"].ToString();
19                 if (sourceValue == null) continue;
20                 var row = dataSource.Values.First()[0];
21                 var column = dataSource.Values.First()[1];
22
23                 var counter = 0;
24                 for (int i = 0; i < reader.FieldCount; i++)
25                 {
26                     if (reader.GetName(i) == "datasource"
27                     || reader.GetName(i) == "ID") continue;
28                     var record = new DataEntry
29                     {
30                         SheetName = sheetName,
31                         Row = row,
32                         Column = column + counter,
33                         Value = reader[i].ToString()
34                     };
35                     counter++;
36                     data.Add(record);
37                     }
38                 }
39             }
40         }
41    }
42    // Build the dictionary structure
```

```
    var result = BuildInternalRepresentation(data);

    return result;
}
```

The BuildInternalRepresentation responsible method for building the internal spreadsheet representation is outlined below. It takes a list of data entries as input. The entries are then placed within the result dictionary, our internal spreadsheet representation. To build the internal spreadsheet it is iterated over the list of data entries. It then checks if the sheet name given in the data entry is already present in the result dictionary as shown in line 8. It creates a new key with the sheet name if it is not present. Afterwards, in line 19 and 20, the value of the data entry is placed within the internal representation according to the coordinates taken from the datasource.

Furthermore, in line 24, the method retrieves the formulas from the Formulas table outlined in section 7.2.1 to build the dedicated formulas sheet. It is then iterated over the formulas as shown in line 28 in order to put the formulas at the according place within the formulas sheet of the internal spreadsheet representation. It can be seen, that if the formulas has a expression defined, the according expression is set instead of the evaluated result.

```
1  static Dictionary<string, string[,]> BuildInternalRepresentation
2  (List<dataEntry> data)
3  {
4      var result = new Dictionary<string, string[,]>();
5
6      foreach (var item in data)
7      {
8          if (!result.ContainsKey(item.SheetName))
9          {
10             var i = data.
11             Where(x => x.SheetName == item.SheetName).Max(x => x.Row);
12             var j = data.
13             Where(x => x.SheetName == item.SheetName)
14             .Max(x => x.Column);
15             result[item.SheetName] = new string[i, j];
16         }
17
18             // Set the values at the specified coordinate
19             and subsequent columns
20             result[item.SheetName][item.Row - 1, item.Column - 1] =
21             item.Value;
22     }
23
24     var formulas = new DbService().GetFormula();
25     var rowMax = formulas.Max(x => x.i);
26     result["Formulas"] = new string[rowMax, 2];
27
28     foreach (var formula in formulas)
29     {
30         result["Formulas"][formula.i - 1, 1] = formula.name;
31         if (string.IsNullOrEmpty(formula.formula))
32         result["Formulas"][formula.i - 1, 0] = formula.result;
33         else
34         result["Formulas"][formula.i - 1, 0] = "=" + formula.formula;
35     }
36
37     return result;
38 }
```

7.2.3 Formula evaluation with HyperFormula

After we built the internal spreadsheet representation, we can use it to create a headless spreadsheet instance which allows for formula evaluation. For this purpose, we use HyperFormula, a JavaScript library that makes use of the headless spreadsheet concept. It includes a formula evaluation engine that allows us to calculate the results of the formulas defined in the internal spreadsheet representation.

To use HyperFormula in our web application, we could execute the respective JavaScript code on the client side. However, as the calculations performed can be resource-intensive, we execute it on the server side. This further allows us to store the calculation result on the database, where it can be retrieved by other users as well.

To execute JavaScript on the server side we can make use of NodeJS [33] which offers a JavaScript runtime environment. To invoke JavaScript on NodeJS from C#, there is a library called Jering which provides an API to do so. For this purpose, a nodeJSService needs to be provided by dependency injection. In the code snippet below, on line 4 it can be seen how the dependency is provided to the Formula Controller. The nodeJSService can be used to invoke JavaScript in an according file as shown on line 14.

```
1  public FormulaController(DbService dbService,
2  INodeJSService nodeJSService)
3  {
4      _nodeJSService = nodeJSService;
5  }
6
7  [HttpGet]
8  public IActionResult CalculateFormulas()
9  {
10     var service = new HeadlessSpreadsheetService();
11
12     var internalRepresentation = service.Process();
13
14     var result = _nodeJSService.InvokeFromFileAsync(
       pathToJavascriptFile, "CalculateFormula",
15     args: new object [] {internalRepresentation });
16
17     return View();
18 }
```

The JavaScript function that is invoked is given below. On line 3 it can be seen how the HyperFormula instance is built by using the buildFromSheets method which takes the internal representation as input. On line 11 it can further be seen it is iterated over the Formulas sheet as long as there is a formula defined. There is the respective formula result and the sheet name read from the HyperFormula instance for each formula. Each formula result and the sheet name is created as a new row and added to the data array as seen from lines 15 to 22. The data array is then returned to the Formula Controller. From there it can be further processed and the results saved to the Formulas table as outlined in section 7.2.1.

```
1  function CalculateFormula(internalRepresentation) {
2
3      const hf = HyperFormula.buildFromSheets(internalRepresentation,
4      {licenseKey: "gpl-v3"});
5
6      const data = [];
7
8      const sheetID = hf.getSheetId('Formulas');
```

```
    var row = 1;

    while (hf.getCellValue(
    {sheet: sheetID, col: 0, row: row }) !== null)
    {
        // Create a new row to add
        const newRow =
        {
        Name: hf.getCellValue(
        { sheet: sheetID, col: 1, row: row }),
        Result: hf.getCellValue({ sheet: sheetID, col: 0, row: row })
        };

        data.push(newRow);
        row++;
    }
    return data;
}
```

8. Correctness testing

In this Chapter, we perform the correctness testing of the headless spreadsheet approach. Software correctness refers to the degree to which a software system meets its specified requirements and performs its intended functions accurately. A correct software system behaves in accordance with its specifications and produces the expected results.

For our correctness testing, we focus on the build process of the internal spreadsheet representation as it is the crucial step of the formula evaluation process as the formula evaluation depends on it. The formula evaluation itself with the headless spreadsheet instance is performed by using a third-party library with its correctness already tested by the provider. Thus, for our correctness testing, emphasis is given on the building of the internal spreadsheet representation.

We perform the tests on the web application presented in Chapter 6. The web application makes use of the headless spreadsheet approach. For the testing of the web application, it must be ensured, that the internal spreadsheet representation is built according to the original spreadsheet. Also, when modifications to the data tables are made within the application and the formula evaluation is triggered, the internal representation needs to reflect the changes made to the data tables. Modifications include Create, Update and Delete operations as summarised below:

- Create: The create operation allows adding new data to a table. This functionality ensures that users can input new data.
- Update: The update operation allows users to modify existing data. This operation ensures that data remains consistent and up-to-date.
- Delete: The delete operation allows that data can be removed when it's no longer required. This is essential for maintaining data.

To verify that the internal spreadsheet representation is built correctly, we perform functional testing of the Create, Update and Delete operations. Functional correctness testing allows us to define test cases and run them to verify that the actual outcome of the test case matches the output we expected beforehand.

It is of great importance to test these operations, as the effectiveness of the headless spreadsheet approach heavily depends on the internal spreadsheet representation since it is required for formula evaluation. In the following sections, we define the test cases, run them and finally analyse the test results.

8.1 Test cases

The test cases are defined such that they allow us to verify that changes applied to the data are considered when it comes to building the internal spreadsheet representation. The primary focus is on ensuring that the internal spreadsheet representation is built correctly, even when modifications to the data have been applied. For this purpose, we defined the test cases in table 8.1. We have defined three test cases which include the Create Update and Delete operations. The test cases allow us to test for the operations if the changes are reflected in the internal spreadsheet representation. The expected outcomes for the test cases are defined accordingly. Defining the test cases beforehand allows us to define an expected outcome for the respective operation. After the tests are executed, we can check if the actual outcome matches the expected outcome.

Operation	Steps	Expected outcome for internal representation
Create	Add data via form	New row is reflected
Update	Update data via form	Updates are reflected
Delete	Delete data	Row is deleted

TABLE 8.1: Test cases

We run the tests by performing the respective operation within the application. For the tests, a simple spreadsheet that includes data about Product Sales is uploaded. We then apply the operations via forms according to the test cases. Afterwards, we trigger the formula evaluation to build the internal spreadsheet representation which we can analyse.

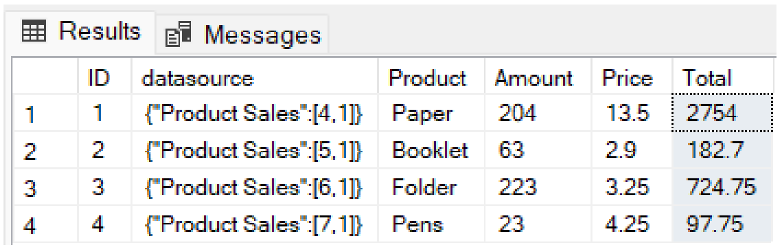
For the test cases we only need to consider valid input, as the input forms for the Update and Create forms do not allow for inputs that do not adhere to the data type of the respective table column. It is therefore not necessary to have a variety of tests with different possible inputs to cover edge cases. This limits the amount of tests that need to be performed to one example of valid data input. The tests of Create, Update and Delete operations cover all types of operations that can have an impact on the internal spreadsheet representation. Thus, by testing these operations we validate the correctness of the internal representation throughout the application's lifetime.

Further, for the correctness testing, it is sufficient to perform single-thread tests. As the Create, Update and Delete operations are performed on the database, the DBMS ensures data consistency and integrity when concurrent operations on the data tables are performed.

8.2 Tests

In the following section, we outline the results of the functional correctness testing. For this purpose, we run the test cases according to table 8.1. For each test case, we prepare an example of the respective operation and assess whether the observed outcome is in accordance with the expected outcome. This includes changes that are applied to data tables are reflected in the internal spreadsheet representation.

As a starting point for our test cases, we consider the Product Sales table as given in Figure 8.1. The table has been imported from a spreadsheet.



Results Messages

	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	2	{"Product Sales":[5,1]}	Booklet	63	2.9	182.7
3	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
4	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75

FIGURE 8.1: Product sales table on database

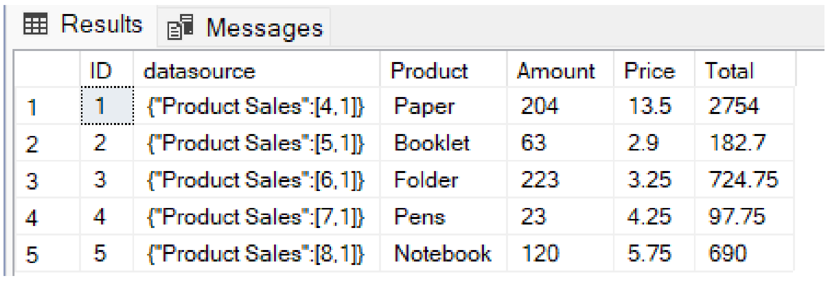
When triggering the formula calculation button on the Formulas View, we obtain The internal representation of the spreadsheet as shown below. It is created as the datasource column indicates.

```
const sheets = {
 'Product Sales': [
   ['Product Sales', '', '', ''],
   ['', '', '', ''],
   ['Product', 'Amount', 'Price', 'Total'],
   ['Paper', '204', '13.5', '2754'],
   ['Booklet', '63', '2.9', '182.7'],
   ['Folder', '223', '3.25', '724.75'],
   ['Pens', '23', '4.25', '97.75'],
  ]
};
```

8.2.1 Create

To test the correctness of the Create operation, we add a new row in the Product Sales table. We can do this with the help of the form for inserting new data. Therefore, we insert a new row into the Product Sales table for the Notebook, which was sold 120 times for a price of 5.75. Thus, the total is 690. We further expect that the internal representation of the spreadsheet includes the newly added row.

After the row has been added to the respective data table, the table looks as shown in Figure 8.2. The new row is added to the table with the according datasource value.



Results Messages

	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	2	{"Product Sales":[5,1]}	Booklet	63	2.9	182.7
3	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
4	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75
5	5	{"Product Sales":[8,1]}	Notebook	120	5.75	690

FIGURE 8.2: Product sales table on database containing new row

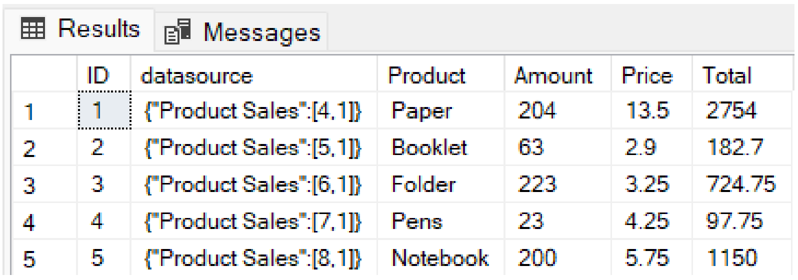
When generating the internal representation based on this updated data table, we obtain the result as shown below. It can be seen that the result is as expected. The newly added row is reflected in the internal representation. It has been added at the end of the table.

```
const sheets = {
 'Product Sales': [
   ['Product Sales', '', '', ''],
   ['', '', '', ''],
   ['Product', 'Amount', 'Price', 'Total'],
   ['Paper', '204', '13.5', '2754'],
   ['Booklet', '63', '2.9', '182.7'],
   ['Folder', '223', '3.25', '724.75'],
   ['Pens', '23', '4.25', '97.75'],
   ['Notebook', '120', '5.75', '690'],
  ]
};
```

8.2.2 Update

To test the correctness of the Update operation, we perform an update of a row in the Product Sales table. This can be done with the help of the update form. Consequently, we update the row that we have previously added. We update the Quantity for the Notebook to 200 and set the Total to 1150 accordingly. We then expect that this update is reflected in the internal spreadsheet representation.

After the row has been updated in the respective data table, the table looks as shown in Figure 8.3. The according row has been updated, whereas the datasource value remains the same.



	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	2	{"Product Sales":[5,1]}	Booklet	63	2.9	182.7
3	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
4	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75
5	5	{"Product Sales":[8,1]}	Notebook	200	5.75	1150

FIGURE 8.3: Product sales table on database containing updated values

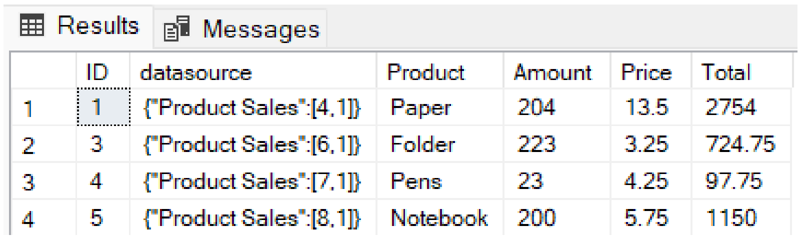
When generating the internal representation based on this updated data table, we obtain the result as shown below. It can be seen that the result is as expected. The updated values are reflected in the internal representation.

```
const sheets = {
 'Product Sales': [
   ['Product Sales', '', '', ''],
   ['', '', '', ''],
   ['Product', 'Amount', 'Price', 'Total'],
   ['Paper', '204', '13.5', '2754'],
   ['Booklet', '63', '2.9', '182.7'],
   ['Folder', '223', '3.25', '724.75'],
   ['Pens', '23', '4.25', '97.75'],
   ['Notebook', '200', '5.75', '1150'],
  ]
};
```

8.2.3 Delete

To test the correctness of the Delete operation, we delete a row in the Product Sales table. This can be done in the Preview of the table. For this purpose, we delete the row of the Booklet. We expect that this row will not be included anymore in the internal spreadsheet representation.

After the row has been deleted in the respective data table, the table looks as shown in Figure 8.4. It can be seen that the according row has been deleted.



	ID	datasource	Product	Amount	Price	Total
1	1	{"Product Sales":[4,1]}	Paper	204	13.5	2754
2	3	{"Product Sales":[6,1]}	Folder	223	3.25	724.75
3	4	{"Product Sales":[7,1]}	Pens	23	4.25	97.75
4	5	{"Product Sales":[8,1]}	Notebook	200	5.75	1150

FIGURE 8.4: Product sales table on database with row deleted

When generating the internal representation based on this data table, we obtain the result as shown below. It can be seen that the result is as expected. The deleted row is now empty in the internal representation.

```
const sheets = {
 'Product Sales': [
   ['Product Sales', '', '', ''],
   ['', '', '', ''],
   ['Product', 'Amount', 'Price', 'Total'],
   ['Paper', '204', '13.5', '2754'],
   ['', '', '', ''],
   ['Folder', '223', '3.25', '724.75'],
   ['Pens', '23', '4.25', '97.75'],
   ['Notebook', '200', '5.75', '1150'],
  ]
};
```

8.3 Conclusion

In conclusion, the results of the functional correctness testing demonstrate the successful validation of the building of the internal spreadsheet representation required for formula evaluation. Specific examples, such as the Create, Update, and Delete operations on the Product Sales table, were illustrated to verify that the actual outcomes are in accordance with the expected outcomes.

Testing functional correctness was structured through the definition of test cases. Each test case focused on a specific operation and specified the expected outcome. The results of the test cases, as illustrated in the subsequent Tests section, confirmed that the internal spreadsheet representation was built as expected and appropriately adhered to changes in the data tables. This ensures the reliability and accuracy of the formula evaluation process within the proposed methodology. It can thus be said that the effectiveness of the developed methodology is given.

9. Performance testing

In the following Chapter, we want to elaborate on the results of performance tests run for the headless spreadsheet approach. The aim is to compare the results to the performance of the same operations performed with the spreadsheet approach. The results are of high importance, as the bad performance of the spreadsheet approach is a key driver of our headless spreadsheet approach. We hereby want to prove that we can deliver better performance with our proposed methodology.

For the headless spreadsheet approach, performance is measured for the main operations of web application presented in Chapter 6. On the other hand, for the spreadsheet approach we prepared an implementation performing the same operations directly on the spreadsheet. This allows us to have an adequate comparison between both the approaches. There are test cases defined according to performed operations for both the approaches. The functionalities which are tested are summarised below:

- Create: The create operation allows adding new data to a table. This functionality ensures that users can input new data.
- Update: The update operation allows users to modify existing data. This operation ensures that data remains consistent and up-to-date.
- Delete: The delete operation allows that data can be removed when it's no longer required. This is essential for maintaining data.
- Read: Read operations which are tested focus on reading results of formula calculations. This operation allows it to get the results provided by the formula of interest.
- Formula evaluation: The formula evaluation operations focus on calculating results for the formulas of interest. This is a critical aspect of the application, as it ensures users can re-evaluate formulas as they make changes to the data.

Our tests include single-thread tests as well as multi-thread tests to simulate multiple users working with the web application. The multi-thread tests allow us to measure how the web application scales with an increasing number of threads.

There are different performance metrics which can be measured for these operations. For our single-thread tests, we focus on the time it takes to process individual operations, i.e. the processing latency. We can perform latency tests on formula evaluation as well as on Create, Read, Update and Delete operations to measure the time it takes to process a request.

For the multi-thread test, we also check the error rates to identify potential issues related to multi-threading and synchronization. These metrics allow us to analyse the responsiveness and efficiency of our web application under different amounts of concurrent users.

For the performance testing, we first outline our test approach. The results of the defined test are then presented. Finally, we analyse the test results.

9.1 Test cases

To test how the performance of the provided operations is affected with the file size in both approaches, we perform the tests on different file sizes. The files used for testing contain a table with 10'000, 100'000 or 200'000 entries. The table contains three columns. Additionally, there is a formula sheet which contains a formula to calculate the average of the values of one of the rows. By testing the different file sizes, we ensure to cover different spreadsheet sizes that could be used in practice. This allows us to test a file with 10,000 entries, which is relatively small, as well as large files with 200,000 entries, which is at the upper end of the range in which it makes sense to manage data in a spreadsheet.

The different operations to be tested on different file sizes lead to a set of test cases. Each test case consists of a test of the runtime approach and our newly proposed methodology. The test cases further differ regarding the operation to be tested and the file size. For each test case, we perform the same action for both approaches. Further, we perform an operation on the table and take a sample size of 250 for each test case to have statistically significant results. This allows us to accurately measure the web applications performance metrics, both on data modification operations as well as formula evaluation and see how it behaves depending on the file size. It is assumed that from a sample size of 250, the effects on the results are minimal if further samples are added. We therefore consider the sample size of 250 to be sufficient.

The described tests for a single thread can be summarised to the test cases as shown below, which are to be tested for both approaches. For each test case, the same operation is performed 250 times per approach to assess the performance of both, the headless spreadsheet and the spreadsheet approaches. The test cases allow us to measure the main operations of the according web applications under different file sizes.

ID	Operation	File size	Data
TCS10k	Create	10'000	Add new data entry
TCS100k	Create	100'000	Add new data entry
TCS200k	Create	200'000	Add new data entry
TRS10k	Read	10'000	Read formula result
TRS100k	Read	100'000	Read formula result
TRS200k	Read	200'000	Read formula result
TUS10k	Update	10'000	Update data entry
TUS100k	Update	100'000	Update data entry
TUS200k	Update	200'000	Update data entry
TDS10k	Delete	10'000	Delete data entry
TDS100k	Delete	100'000	Delete data entry
TDS200k	Delete	200'000	Delete data entry

TABLE 9.1: Single-thread test cases

In addition to the single-thread tests, we perform multi-thread tests to simulate users who perform operations at the same time. Performance test results for single-threaded operations provide a baseline for our application's performance. However, real-world scenarios often involve multiple concurrent users or threads. Thus, to get adequate results, we test both, single and multi-thread scenarios. The test cases for multi-thread tests are given in the table below. The focus is on Update operations, as they are the most important operation to be tested in a multi-threaded scenario where multiple users can work on the same data concurrently.

ID	Operation	File size	Threads	Data
TUM10k5	Update	10'000	5	Update data entry
TUM10k10	Update	10'000	10	Update data entry
TUM10k100	Update	10'000	100	Update data entry
TUM4100k5	Update	100'000	5	Update data entry
TUM100k10	Update	100'000	10	Update data entry
TUM100k100	Update	100'000	100	Update data entry

TABLE 9.2: Multi-thread test cases

For formula evaluation, we have the test cases as shown in table 9.3. With these test cases, we want to test how long it takes to re-evaluate formulas after changes have been made to the data. For this purpose, we only do single-thread tests as it is sufficient if the re-evaluation is only performed by a single thread as the result is stored in the database and can be retrieved from there without having to re-evaluate the formulas.

ID	Operation	File size
TE10k	Formula evaluation	10'000
TE100k	Formula evaluation	100'000
TE200k	Formula evaluation	200'000

TABLE 9.3: Formula evaluation test cases

To assess the performance of the two approaches, we have the according functionalities prepared in the web application as outlined in 6 for the headless spreadsheet approach as well as a similar implementation for the spreadsheet approach. This allows for the evaluation of performance by sending HTTP requests to the web applications for which we can measure the performance. The tests are performed on CRUD operations (Create, Read, Update, Delete) on the data tables as well as on formula re-evaluation, which is required when the data on which formulas are evaluated has changed.

All the performance tests are performed under the same conditions. The tests are performed locally on a system which has an i7 CPU 2.8 Ghz, 32 GB RAM, Win11, and a Samsung mzvlb1t0hblr-000l7 SSD drive. The tests are performed within JMeter by creating a thread group, the corresponding sampler, configuration files and listeners. As the tests are performed locally, we do not expect network latency, instead we obtain the processing latency for the tested operations.

9.2 Single-thread tests

In the following section, we outline the single-thread tests. The results of the performance tests according to the defined test cases in table 9.1 for the Create, Read, Update and Delete operations of a single thread are given. The tests are performed on both the headless spreadsheet and spreadsheet approaches.

To test the latency of CRUD operations in JMeter we need to set up an Http Post request accordingly. It needs to be a post request, as we send the data to be added to the body of the request.

When performing single-thread tests for CRUD operations of the headless spreadsheet approach, we get similar results for each operation. As the operations can be performed on the database, we get an average latency of 0-4 milliseconds for each operation. Also, the standard deviation lies within a few milliseconds.

On the other hand, we get the results for latency as shown in table 9.4 when performing the same operations with the spreadsheet approach:

Test case ID	Average	Min	Max	Std. Dev.
TCS10k	177	142	444	25
TCS100k	1734	1429	2190	90
TUS200k	3315	2432	4123	118
TRS10k	117	72	165	18
TRS100k	750	260	992	134
TRS200k	1700	1592	2231	131
TUS10k	170	135	418	23
TUS100k	1771	1538	2281	92
TUS200k	3305	2659	4093	128
TDS10k	240	160	310	37
TDS100k	1063	657	1299	114
TDS200k	2200	1145	2534	220

TABLE 9.4: Latency results in ms for spreadsheet approach

9.3 Multi-thread tests

In the following section, we perform the multi-thread tests. The results of the performance tests according to the defined test cases in table 9.2 for the Read and Update operations of a multi-thread are given.

The multi-thread tests are performed only on the headless spreadsheet approach. The reason is that for the spreadsheet approach, concurrent modifications on local spreadsheets are not possible due to file locking. They are only possible when providing it on a shared drive. There are according APIs, such as the Microsoft Graph API [26] for Excel, to allow collaboration of multiple users. However, tests have shown that such APIs cause bad performance as the network latency to access the provider's server is much higher than when using a library to work with a spreadsheet locally as outlined in 2.4. Also, when using such APIs, the access can be limited by a throttling mechanism implemented by the respective server [25]. We therefore concluded that for the spreadsheet approach, it does not make much sense to perform any multi-thread tests. Instead, we focus on the tests of the compile time approach.

Regarding the multi-thread test cases, we perform the same latency and load test as in the single-thread test, with the minor adjustment that we increase the number of threads to see how an increasing number of concurrent users influences the results depending on the file size.

In Table 9.5, we outline the test results of the latency of the Update operations for multi-threads. This includes test cases according to Table 9.2.

Test case ID	Threads	Average	Min	Max	Std. Dev.	Error rate
TUM10k5	5	1	0	20	1	0%
TUM10k10	10	2	0	66	4	0%
TUM10k100	100	24	1	373	15	0%
TUM100k5	5	1	0	50	3	0%
TUM100k10	10	2	0	47	3	0%
TUM100k100	100	25	0	332	15	0%

TABLE 9.5: Latency results in ms for headless spreadsheet approach

9.4 Formula evaluation

In the following section, we test the latency of the formula evaluation. This includes test cases as outlined in Table 9.3. We perform the tests for both the headless spreadsheet and the spreadsheet approaches to compare the results.

To re-evaluate the formulas in the spreadsheet approach, it is required to trigger a recalculation of the spreadsheet. On the other hand, for the headless spreadsheet approach, it is required to build a headless spreadsheet instance and read the results from it. Thus, the formula evaluation process includes building the internal spreadsheet representation which is required for the headless spreadsheet instance.

For the test cases, a spreadsheet is used that contains a formula with which the sum of one of the columns of the table is calculated, which has a size corresponding to the defined test case. For the tests, this formula is re-evaluated. In table 9.6, the results for the runtime approach are given.

Test case ID	Average	Min	Max	Std. Dev.
TE10k	208	142	265	25
TE100k	1391	231	2342	60
TE200k	2105	1574	2225	82

TABLE 9.6: Latency results in ms for spreadsheet approach

In table 9.7, the results for the headless spreadsheet approach are given.

Test case ID	Average	Min	Max	Std. Dev.
TE1	142	80	333	50
TE2	1192	820	1758	178
TE3	2340	1575	2936	272

TABLE 9.7: Latency results in ms for headless spreadsheet approach

9.5 Analysis of test results

When analysing the single-thread test results of the spreadsheet approach, it becomes clear that for the Create and Update operations, we obtain approximately the same results as technically, the operations performed are very similar. In both cases, new values are written in a spreadsheet. In the case of an update, an old value is just overwritten. Regarding latency, it can be seen, that depending on file size, the operation can take up to a few seconds. The larger the file, the larger the latency. The results of the performance tests behave linearly depending on the file size. It can also be seen that Create, Update and Delete operations are especially expensive, whereas read operations can be performed faster.

On the other side, for the headless spreadsheet approach, all CRUD operations can be performed on the database which can be done within a few milliseconds, not depending on the file size. For the single-thread tests, it can be seen that for all test cases, we can provide better performance results with our headless spreadsheet approach. The reason is that we don't have to load the spreadsheet at runtime to change values. Instead, we can make changes to the database and profit from performance optimizations implemented in a DBMS.

Regarding formula evaluation, it can be seen that for both the approaches, re-evaluation can take up to a few seconds depending on the file size. In the spreadsheet approach, the spreadsheet is loaded and used for formula calculation. On the other hand, in the headless spreadsheet approach the formula evaluation is equally expensive as we have to build our internal spreadsheet representation and the headless spreadsheet instance for formula evaluation.

When analysing the multi-thread tests on the headless spreadsheet approach, it can be concluded that we can use the approach to be used by multiple users. It can be seen that the latency for CRUD operations increases as the number of threads increases, but does not depend on file size. Still, the latency for the update operation was 24-25 milliseconds for 100 threads operating simultaneously. Thus, the results indicate that the approach is suited to be used by multiple users at the same time. Also, it can be seen that the error rate was 0% for every test case, indicating that there are no issues related to multi-threading and synchronisation.

The outcomes of the test result show that we have significantly improved performance with the headless spreadsheet approach compared to the spreadsheet approach for the CRUD operation. It can be seen that it is more efficient to work with an internal representation of a spreadsheet than to work with the spreadsheet application at runtime, thus having to load the entire workbook for every request. When using the headless spreadsheet approach we can benefit we can profit from less overhead compared to the spreadsheet approach and performance-increasing techniques such as the AST and dependency graphs.

However, for performance test results, it must also be noted that the outcomes heavily depend on the infrastructure on which the web application is hosted. As we have intense I/O database operations, a potential performance bottleneck can be disk I/O speed. For example, tests have shown that when performing the tests with an HDD instead of a SDD, the performance results are significantly worse. Furthermore, if the web application is hosted in a productive environment, network latency is also to be considered. When performing the tests locally, this is not a factor to be considered. Further, the results of the performance test could improved by providing a server with better CPU power and more cores, which is especially important for the muli-thread test. Thus, it must be noted that the server infrastructure on which the web application is hosted has a strong impact on the performance.

10. Limitations and future work

The proposed headless spreadsheet approach and its implementation in the web application, as described in 6, have some limitations, which need to be explained in detail. It also needs to be outlined what future work needs to be done to further refine the proposed headless spreadsheet approach and its implementation.

In the following Chapter, we outline the limitations and future work for our proposed headless spreadsheet approach. Both stages of the approach have their own limitations and future work that needs to be done. Thus we first elaborate on the limitations and future work of the transfer of data from the spreadsheet to the database during the initial upload of the spreadsheet as outlined in Chapter 4. Afterwards, we do the same for the formula evaluation stage as outlined in Chapter 5.

10.1 Transfer of data from spreadsheet to database

In the following section, we elaborate on the limitations and future work of the transfer of data from the spreadsheet to the database during the initial upload of the spreadsheet as outlined in Chapter 4. The according limitations and future work are given below:

- The headless spreadsheet approach does not implement a method for detecting primary keys and foreign keys. It is clear, that a methodology, which is able to correctly identify keys would be much more powerful.
- Missing values in tables of the spreadsheet document to be uploaded need to be replaced with either "N/A" or "n/a". The headless spreadsheet approach does currently not provide the possibility to automatically handle missing values correctly.
- Rule-based approaches for table detection and table recognition have the inherent limitation, that they can only correctly detect and recognize tables that follow a certain structure. Tables that might follow a different structure, cannot be correctly identified. A more automated approach, i.e. based on Machine Learning techniques could allow for a better success rate in correctly identifying tables.
- As the headless spreadsheet approach only considers the existence of one internal representation of a spreadsheet, it is currently not possible to consolidate several spreadsheets within the same application. For, use cases where this would be required, the approach needs to be adapted accordingly.

- As in the correctness testing performed in Chapter 8, the focus is on the building of the internal spreadsheet representation, we did not perform a formal testing of the proposed table detection and recognition algorithm. In practice, it has shown to work out well, however there is further in-depth testing required. Also, it would be required to perform an integration test of the SQL statements generated and their execution on the DBMS.
- Our proposed implementation, uses EPPlus to read from the original spreadsheet during table detection and recognition. As EPPlus only allows to work with Microsoft Excel, our current implementation does not allow to work with other spreadsheet software. To allow it to work with other spreadsheet software as well, the implementation must be changed to work with a different technology than EPPlus.

10.2 Formula evaluation

There are also limitations for the second stage of the headless spreadsheet approach which is the formula evaluation as outlined in Chapter 5. The according limitations and future work are given below:

- For the headless spreadsheet approach to work, it is required that the spreadsheets are prepared as outlined in section 5.4. It is currently not possible to work with any kind of spreadsheet. Thus, it is always required to put effort into preparing the spreadsheet accordingly and follow the according structure that must be given within the spreadsheet. This includes the setup of the data tables such that they can be correctly detected and recognised as well as setting up the Formulas Sheet and defining the formula there.
- It is also clear that the headless spreadsheet approach is not suitable for all use cases. It is well suited for use cases where it is not important that the results of the formula calculation are available in real time, or when working with rather small spreadsheets. However, the headless spreadsheet approach is not suitable for use cases in which large spreadsheets are used and the data changes every few seconds and the calculation results must be available in real-time, as it can take several seconds to recalculate the formulas in this case.
- The performed correctness testing as outlined in Chapter 8 assumes only valid user input can be given as there is a front-end validation of input. However, there is no backend testing performed which also includes invalid user input and edge cases. To test the correctness of the approach on the backend, additional tests need to be performed.
- In order for the formula evaluation to work properly throughout the application's lifetime, the tables in the original worksheet must not be arranged one below the other. Otherwise, when data is added to the upper table, the datasource of the new data can possibly conflict with the datasource of the table below. To avoid this, it is the user's responsibility to arrange the tables accordingly.

11. Summary

We propose an approach which allows it to use spreadsheet documents as requirements specification for automatic software generation. The idea is to have the data from the spreadsheet available in the generated application and to be able to use it to evaluate formulas defined in the original spreadsheet. We call this approach the "headless spreadsheet approach". Compared to the existing spreadsheet approach [18], the headless spreadsheet approach aims to work independently of the spreadsheet document. The spreadsheet is only required during the initial upload.

To come up with the headless spreadsheet approach, we conducted constructive research where we created a methodology. The headless spreadsheet approach aims to answer the first two research questions which are about how to transfer the data from the spreadsheet to the database and how to evaluate the formula based on this data. For this purpose, there are two stages of the headless spreadsheet approach.

The first stage is about the transfer of tables from spreadsheet documents to a relational database. This stage can be split into the following steps: table detection, table recognition, creation of a data model and storing the tables in a relational database. With this stage of the headless spreadsheet approach, we answer the first research question.

Table detection is the process of correctly identifying the dimensions of all tables in an Excel document. On the other hand, table recognition means the process of correctly assigning a cell to a predefined cell type. For Both processes, we have implemented the rule-based approach proposed by Doush and Pontelli [7]. Table detection and table recognition are combined within the same algorithm. After tables have been correctly detected and recognized, there can be a data model created accordingly. Subsequently, this allows us to create the corresponding database schema and insert the data into the database.

The second stage of our headless spreadsheet approach is to be able to evaluate formulas defined in the original spreadsheet in the generated application. To do so, we make use of the concept of headless spreadsheets [13]. Headless spreadsheets allow it to evaluate spreadsheet formulas when they are given the sheets as a two-dimensional array as input. To provide the sheet data, we create an internal representation of the spreadsheet based on the data that has been transferred from the spreadsheet to the database.

To build an internal spreadsheet representation that resembles the structure of the original spreadsheet, we need information on the data tables available which can be used to know the location of the data in the original spreadsheet. For this purpose, each data table has a "datasource" column, providing the information about at which sheet and at which coordinates the data needs to be located to reconstruct the original spreadsheet. This allows it to correctly build the internal spreadsheet representation which we can further use to create the headless spreadsheet instance required for formula evaluation.

To demonstrate how the headless spreadsheet approach could work out in practice, we have created a web application that makes use of the approach. On the web application, it is possible to upload a prepared spreadsheet. Within the generated functionality, the user can then work on this data and evaluate formulas based on the data. Upon the user's request to evaluate the formulas, the internal spreadsheet representation and the according headless spreadsheet instance are created. This allows the user to evaluate the formulas defined in the original spreadsheet.

We further have tested the correctness of the headless spreadsheet approach to answer the third research question about the correctness of the elaborated approach. At the testing, we focused on the process of building the internal spreadsheet representation. The testing allowed us to verify that the internal spreadsheet representation is built correctly, even when changes to the data have been made within the application.

Our performance tests have finally shown that the headless spreadsheet approach offers better performance results than the existing spreadsheet approach which depends on the spreadsheet at runtime. The results, allow us to answer the fourth research question which is about comparing the performance results of the two approaches.

Bibliography

[1] E. Aivaloglou, D. Hoepelman, and F. Hermans, "Parsing excel formulas: A grammar and its application on 4 large datasets: Aivaloglou et al .," *Journal of Software: Evolution and Process*, vol. 29, e1895, Sep. 2017. DOI: 10.1002/smr.1895.

[2] M. Alexander, *Excel 2019 Bible / Michael Alexander, Dick Kusleika ; previously by John Walkenbach.* eng, 1st edition. Indianapolis, Indiana: Wiley, 2019, ISBN: 1-119-51476-2.

[3] N. Astrakhantsev, D. Turdakov, and N. Vassilieva, "Semi-automatic data extraction from tables," in *RCDL*, 2013.

[4] N. Bencomo, S. Götz, and H. Song, "Models@ run. time: A guided tour of the state of the art and research challenges," *Software & Systems Modeling*, vol. 18, pp. 3049–3082, 2019.

[5] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Sep. 2012, vol. 1. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[6] J. Cunha, J. Saraiva, and J. Visser, "From spreadsheets to relational databases and back," in *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '09, Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 179–188, ISBN: 9781605583273. DOI: 10.1145/1480945.1480972. [Online]. Available: https://doi.org/10.1145/1480945.1480972.

[7] I. A. Doush and E. Pontelli, "Detecting and recognizing tables in spreadsheets," in *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, ser. DAS '10, Boston, Massachusetts, USA: Association for Computing Machinery, 2010, pp. 471–478, ISBN: 9781605587738. DOI: 10.1145/1815330.1815391. [Online]. Available: https://doi.org/10.1145/1815330.1815391.

[8] EPPlus Software AB. "Epplus - features and technical overview." (May 2023), [Online]. Available: https://www.epplussoftware.com/fr/Developers/FormulaCalc.

[9] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007. DOI: 10.1109/TSE.2007.70731.

[10] Gartner. "What are analytics and business intelligence platforms?" (Apr. 2023), [Online]. Available: https://www.gartner.com/reviews/market/analytics-business-intelligence-platforms.

[11] Gartner. “What is an enterprise low-code application platforms?” (Apr. 2023), [Online]. Available: https://www.gartner.com/reviews/market/enterprise-low-code-application-platform.

[12] Handsoncode. “Dependency graph.” (Jun. 2023), [Online]. Available: https://hyperformula.handsontable.com/guide/dependency-graph.html#cells-in-the-dependency-graph.

[13] Handsoncode. “Hyperformula guide.” (May 2023), [Online]. Available: https://hyperformula.handsontable.com/#what-is-hyperformula.

[14] Handsoncode. “Hyperformula performance.” (Jul. 2023), [Online]. Available: https://hyperformula.handsontable.com/guide/performance.html#suspending-automatic-recalculations.

[15] B. Held, B. Moriarty, and T. Richardson, *Microsoft Excel Functions and Formulas with Excel 2019/Office 365*, eng, 5th edition. Boston, Massachusetts: Mercury Learning and Information, 2019, ISBN: 9781683923749.

[16] F. Hermans, B. Jansen, S. Roy, E. Aivaloglou, A. Swidan, and D. Hoepelman, “Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 56–65. DOI: 10.1109/SANER.2016.86.

[17] E. Koci, M. Thiele, O. Romero, and W. Lehner, “A machine learning approach for layout inference in spreadsheets,” in *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, ser. IC3K 2016, Porto, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 77–88, ISBN: 9789897582035. DOI: 10.5220/0006052200770088. [Online]. Available: https://doi.org/10.5220/00060522%2000770088.

[18] M. Wahler and E. Conte and M. Frick and J.D. Mosquera Tobon and M. Ruiz, “Boosting business agility with model-driven engineering,” pp. 1–12, 2022.

[19] Martin Frick. “Using spreadsheet documents as requirements specifications for automatic software generation.” (Jan. 2024), [Online]. Available: https://github.zhaw.ch/artifact-driven-engineering/excel-hyperformula-msc-martin-frick.

[20] A. Mazurkiewicz, “Introduction to trace theory,” Mar. 1995. DOI: 10.1142/9789814261456_0001.

[21] Microsoft-Corporation. “Built-in reference types (c reference)).” (Oct. 2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types.

[22] Microsoft-Corporation. “Bulk insert (transact-sql).” (Dec. 2023), [Online]. Available: learn.microsoft.com/en-us/sql/t-sql/statements/bulk-insert-transact-sql?view=sql-server-ver16.

[23] Microsoft-Corporation. “Create a database schema).” (Oct. 2023), [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/authentication-access/create-a-database-schema?view=sql-server-ver16.

[24] Microsoft-Corporation. “Data types (transact-sql).” (Oct. 2023), [Online]. Available: https://learn.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql?view=sql-server-ver16.

[25] Microsoft-Corporation. "Microsoft graph throttling." (Dec. 2023), [Online]. Available: https://learn.microsoft.com/en-us/graph/throttling.

[26] Microsoft-Corporation. "Overview of microsoft graph." (Dec. 2023), [Online]. Available: https://docs.microsoft.com/en-us/graph/overview.

[27] Microsoft-Corporation. "Refer to cells and ranges by using a1 notation." (May 2023), [Online]. Available: https://learn.microsoft.com/en-us/office/vba/excel/concepts/cells-and-ranges/refer-to-cells-and-ranges-by-using-a1-notation.

[28] Microsoft-Corporation. "Sqlconnection class)." (Oct. 2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection?view=dotnet-plat-ext-7.0.

[29] Microsoft-Corporation. "Sqlconnection.connectionstring property)." (Oct. 2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring?view=dotnet-plat-ext-7.0.

[30] Microsoft-Corporation. "Overview of asp.net core mvc." (Jan. 2024), [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0.

[31] Microsoft-Corporation. "Sqlcommand class." (Jan. 2024), [Online]. Available: hhttps://learn.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=dotnet-plat-ext-8.0.

[32] Microsoft-Corporation. "Transact-sql statements." (Jan. 2024), [Online]. Available: https://learn.microsoft.com/en-us/sql/t-sql/statements/statements?view=sql-server-ver16.

[33] OpenJS Foundation. "Introduction to node.js." (Jan. 2024), [Online]. Available: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs.

[34] Softr. "Softr." (Apr. 2023), [Online]. Available: https://www.softr.io/.

[35] K. Taylor, "An analysis of computer use across 95 organisations in europe, north america and australasia," *Wellnomics, Christchurch, New Zealand*, 2007.

[36] The apache software foundation. "Formula evaluation." (May 2023), [Online]. Available: https://poi.apache.org/components/spreadsheet/eval.html.

[37] The International Business Machines Corporation (IBM). "Identifying keys and analyzing relationships." (Oct. 2023), [Online]. Available: https://www.ibm.com/docs/en/iis/9.1?topic=relationships-primary-key-analysis.

[38] X. Wang, "Tabular abstraction, editing, and formatting," Sep. 2000.

[39] Willem, Jann. "Xlparser." (May 2023), [Online]. Available: https://github.com/spread%20sheetlab/XLParser.

[40] Y. Zhao and F. Rammig, "Model-based runtime verification framework," *Electronic Notes in Theoretical Computer Science*, vol. 253, pp. 179–193, Oct. 2009. DOI: 10.1016/j.entcs.2009.09.035.