Transferring heterogeneous data from generic databases into a SQL database using HTTP
Possibilities and Implementation

Jesper Bilander

Master’s thesis in computing science, 30 ECTS credits. Master of Science Programme in Computing Science and Engineering, 300 ECTS Credits. In collaboration with Trimma AB.

Trimma mentor: Adam Holmström
CS-UmU Supervisor: Jan-Erik Moström
Examiner: Henrik Björklund
Abstract
The thesis investigates the possibility to create software that can synchronize heterogeneous data from generic databases through the Internet. To further enhance the conclusion, we implement a prototype that fetches data from numerous source system databases, and stores the data in another location, more accessible for processing. To achieve that, previous work with topics related to the area was inspected and analyzed. Based on the investigation, the implemented prototype uses a client-server approach which communicates with a REST-like design using JSON-strings and HTTP. This thesis and the resulting prototype proves that it is fully possible to create such software by combining and using existing protocols and frameworks.
Acknowledgements

I would like to show gratitude to my supervisor, Jan-Erik Moström, for his guidance and feedback throughout the project. Secondly, I would like to thank Trimma AB for welcoming me as one of them, and for lending me a workstation. A special note to Andreas Åström for giving me the opportunity to perform the master thesis project in collaboration with them. I also want to give an acknowledgment to Adam Holmström, my mentor at Trimma, for his counseling and interest during the development of the prototype.
# Contents

## 1 Introduction

- 1.1 Trimma .................................................. 1
- 1.2 Motivation .................................................. 1
- 1.3 Problems ................................................... 2
- 1.4 Methodology ............................................... 4
- 1.5 Thesis organization ....................................... 4

## 2 Investigation

- 2.1 Heterogeneous SQL synchronization .......................... 8
- 2.2 Transferring data between heterogeneous databases using XML 8
- 2.3 Possibilities of heterogeneous database types .................. 10
- 2.4 JSON and XML .............................................. 10
- 2.5 CORBA ..................................................... 11
- 2.6 SOAP and REST ........................................... 12
- 2.7 Tunneling SQL through HTTP ................................. 13

## 3 Analysis of investigation

- 3.1 Transferring and representing the data ......................... 14
- 3.2 Database synchronization and change tracking .................. 15
- 3.3 Heterogeneous databases ..................................... 15

## 4 Resulting prototype

## 5 Implementation

- 5.1 Collecting the data .......................................... 19
- 5.2 Transferring the data ........................................ 20
- 5.3 Storing the data ............................................. 20
- 5.4 Implementing support for another database type ................ 20

## 6 Prototype evaluation

## 7 Discussion

- 7.1 Deviations from the project plan ................................ 23
- 7.2 Further improvements .......................................... 24
  - 7.2.1 Security .................................................. 24
  - 7.2.2 Threaded server program ................................ 24
  - 7.2.3 Change tracking .......................................... 25
  - 7.2.4 Insert optimization ...................................... 25
  - 7.2.5 Graphical configuration interface ......................... 26

## 8 Conclusion

## References
List of Tables

2.1 Comparison between JSON and XML . . . . . . . . . . . . . 11
D.1 Transfer API of the prototype. . . . . . . . . . . . . . . . . 32
List of Figures

1.1 On-premises system overview ........................................ 2
1.2 Cloud system overview .................................................. 2
4.1 Simplified flowchart of the prototype .............................. 17
4.2 Overview of prototype .................................................. 18
B.1 Configuration tables ..................................................... 30
Nomenclature

Specific words, acronyms and abbreviations used throughout the report that might not have an apparent meaning.

Words

.NET Framework A software framework developed by Microsoft, primarily for Microsoft Windows.

SQL-Server A relational database management system by Microsoft, works with SQL.

Acronyms & Abbreviations

API Application Programming Interface. Defined methods of communication between software components.

CORBA Common Object Request Broker Architecture. A protocol for distributed systems communication.

JSON JavaScript Object Notation. A way to represent data structures that focuses on readability for humans.

REST Representational State Transfer. A data transfer protocol.


SQL Structured Query Language. Used to communicate with SQL-databases.

VPN Virtual Private Network. A method to allow computers using a public network to access a private network.

XML Extensible Markup Language. A way to represent data structures, in a human readable form.
1 Introduction

Trimma has requested a way to use data from their customers’ source system databases without the need of installing their system on the customers’ hardware, or using VPN-tunnels. The data from the customer’s database is used in Trimma’s decision support system for company management. To easier maintain and use the system in a cloud environment it must be separated from the source system. Our part in the separation process is to find and create a convenient way to solve the data transfer. Since each source system is different, they potentially have very different storage solutions. Heterogeneity is therefore an obstacle needed to be taken care of. An important side note for this thesis is that the resulting prototype is a proof-of-concept, rather than of enterprise grade.

1.1 Trimma

Trimma is an IT-company established in 2003 and is the developer of the product INSIKT. INSIKT is a web-based front-end platform utilizing Microsoft technology. The product is a market leading tool for having a company’s analytic, reporting, planning, budgeting and score-carding in one and the same integrated solution.

1.2 Motivation

INSIKT need information from the company’s databases to operate. Trimma’s current go-to solution for this is to install the system as an on-premises software. This means that it has to be uniquely adapted for each source system. This system is illustrated in [Figure 1.1]. The intention is therefore to create a prototype which enables a system as illustrated in [Figure 1.2]. Thus, the motivation for this thesis is to investigate and create a generic way of gathering, transferring, and storing data from different databases.
1.3 Problems

The project in this thesis have three different problem areas, gathering, transferring, and storing the data.

1.3.1 Gathering the data

The diversity of the source systems is the primary problem. Thus, the solution needs to be adaptive to the database type and the data in the source system. The time that the source system’s database is locked needs to be kept to a minimum. Another aspect that needs elucidation is the possibility to ignore already transferred data during the gathering process.

1.3.2 Transferring the data

Choosing the best-suited transfer method is essential. Transfers typically occur once a day, so there is no real performance threshold in the form of
transfer speed. However, the solutions must be able to transfer a considerable amount of data.

1.3.3 Storing the data

It is crucial that the data is mapped and arranged appropriately to fit the target database’s structure.
1.4 Methodology

The procedure of the master’s thesis is described by the sections below. In addition, the work process for each section is summarized.

1.4.1 Investigation

In the beginning, we assembled previous work and information from search engines such as IEEE’s digital library. The goal was to find information within the same area as the thesis. Thus, creating a base that could aid and influence future design decisions. To get inspiration with problems during the actual implementation of the prototype. We also gathered information from more traditional search engines such as Google.

1.4.2 Analysis

At this point, we analyzed and evaluated the investigation material based on the prototype’s needs and the problems stated in section 1.3. The analysis compares similar approaches to a given problem and estimates what is most suitable for the prototype.

1.4.3 Implementation

A prototype was implemented based on the analysis. We made the decision to write it in C# using Visual Studio because that is the environment that Trimma uses.

1.4.4 Evaluation of resulting prototype

After all of the previous stages were finished, we analyzed the prototype. This was done by testing the prototype with different databases and see how it functioned. What was easy/hard to understand when using it? Could we find specific cases where the prototype had problems to operate as planned? The first step was to describe the resulting prototype. The second was to perform tests and describe the results. Lastly, the third step was to discuss the material created from the earlier steps.

1.5 Thesis organization

1. Introduction
   Explains the motivation, purpose, and methodology of the thesis.
2. Investigation
   Contains material from previous work that treats the same area as the thesis.

3. Analysis of investigation
   An analysis of the information gained from the investigation is held, it is evaluated based on the needs of the prototype.

4. Resulting prototype
   The prototype that is the result of the evaluation and implementation is presented.

5. Implementation
   Explains the implementation of the prototype as well as how to modify it for a new source system.

6. Prototype evaluation
   Contains the material that arose from testing and evaluating the prototype.

7. Discussion
   A discussion about the resulting prototype, possible other approaches, and further development is held.

8. Conclusion
   Concludes the thesis. Connects the purpose and discussion together with the result.

9. References
   A list with the references used while writing the report.
2 Investigation

The design of the prototype is based on the material presented in this section. The section starts with an introduction of the thought process during the investigation together with a short description of the material. After the introduction, all material is presented. The investigation is within a somewhat unexplored area since it is hard to find definite answers or work on the problem as a whole.

To be able to create a good design it has been important for us to stay unbiased and receptive. We have therefore made a broad sweep of available technologies before deciding on our solution. What follows is an in-depth description of our findings, followed by an analysis in Section 3.

The investigation is divided in two parts, gathering and storing, and transferring the data. However, the articles are not presented in this manner, as a few of them contained information within both categories.

During the gathering and storing investigation, we focused on articles containing heterogeneous database synchronization. This led to the material presented in two sections:

- **Section 2.1** - describes a solution which copies and remolds the queries from the source database to fit the target database as well.
- **Section 2.2** - describes a solution which uses an XML middle-tier representation of the data to synchronize and map it between the databases.

The presented material is helpful, but it does not focus on being generic for the source database. Due to this we also included a less scientific source, **Section 2.3**, which proposes a solution to that problem.
We focused on two areas during the **transferring** investigation. How to represent the data, and how to transfer it. As a data representation, we wanted something we were familiar with and well-known in the community. The choice was to investigate JSON and XML, and for these, it was sufficient to use a small summary. However, we purposely found a biased article for each method to further strengthen the decision. This material is presented in [Section 2.4](#). With this in mind, we searched for ways to transfer the data. This brings us to SOAP and REST which are two well-known ways for distributed system communication. We also wanted to investigate less traditional methods of using HTTP, and other ways of performing distributed system communication. This was done to get inspiration and open our minds to other solutions or adaptations. All this material is presented in three sections:

- **Section 2.5** - describes CORBA, one of the first tools of creating communication within distributed systems.
- **Section 2.6** - discusses the difference between SOAP and REST as well as giving an insight of things to consider when choosing between the two.
- **Section 2.7** - describes a system that tunnels SQL commands through HTTP to bypass the database native communication port.

We were also introduced to other methods which sounded very appealing to use. These were Google protocol buffer[^1] and Apache Thrift[^2]. However, neither of them is a solution proper for the prototype. Google only recommend trying the new proto3 version (which supports C#) of their protocol buffers if you wanted to try using protocol buffers in one of the newly-supported languages. This meant that the method was too risky to use, as they do not recommend to use it in important projects. Apache, on the other hand, stood behind their product with more confidence. Nonetheless, running into unnecessary trouble felt inevitable, as the compiler looked problematic to install. In conclusion, it felt safer to use something that had been around for a while, and in addition had a shorter startup process.

[^1]: https://developers.google.com/protocol-buffers/docs/overview
[^2]: http://thrift.apache.org/
2.1 Heterogeneous SQL synchronization

This section is based on the article "SQL query based data synchronization in heterogeneous database environment" by Md. Iqbal Hossain and Muhammad Masroor Ali. Since their work closely relates to the task at hand, we will be able to use their work to get a greater understanding of the problem, and a possible solution as a whole. They achieve Unidirectional synchronization by detecting the SQL query in the source database and passing it to the remote target database through a HTTP protocol. The main advantage of this method is that it does not process all data of the source database.

One of the problems with heterogeneous database systems is the possibility of data type mismatch or unavailability. To avoid this, it is possible to use look-up tables and adapt the SQL-query based on them, before executing the query in the target database. The solution has a table for each source database. This is not needed if the source database and target database uses the same data types. After having adapted the query based on the look-up table, the query is executed. If successfully executed, it will save the query in a text file with archived queries. If not successfully executed it will save the error in an error log file and stop executing the following queries, as the queries need to be executed in the same chronic order to achieve synchronization.

Both WisdomForce and Oracle have systems for database synchronization. However, WisdomForce’s DatabaseSync is very costly, and it can only do data synchronization. It can not detect any change in schema of source database. For example, if any new table is created in the source database, WisdomForce’s DatabaseSync can not automatically create this table in the corresponding schema of the target database. Oracle Active Data Guard only supports Oracle to Oracle databases and thus, can not be implemented in a heterogeneous system because of it.

2.2 Transferring data between heterogeneous databases using XML

This section is based on the master thesis "XML data transfer between heterogeneous database systems" by Qinghong Lin. The thesis focuses on a situation dealing with transferring and sharing data between two data sources with different data models that are based on the same real world object. He designs and implements a tool for data transfer between heterogeneous schemas by utilizing XML technologies and using the Microsoft .NET development platform. This thesis is included to get information on how XML can be used as a data representation when transferring data in the .Net framework. It is also interesting in the point of transferring generic data.

Databases are used to represent data models, which reflects real world objects and their relationship with each other. Since real world problems are analyzed by different groups of people, it is likely to produce different results when representing them in databases. Imagine two databases dealing with the same real world case, where each schema can name their tables and columns differently. They can also have different relations, key constraints, data types and cases where two attributes
corresponds to one attribute in the other schema. Having these miss-matches brings problems when sharing and reusing the information. It is thus important to find a way to match between the schemas, that is efficient and easy to process.

XML is about describing and formatting data. It is a reliable, flexible, and broad structuring standard. By defining a generic syntax for marking data with simple, human-readable tags, it can be completely customized to fit different data representations. XML also shares some attributes with objects in object-oriented programming, element content, subtypes, named attributes, and hierarchical structure representation. Even with these expressive features, it is light, easy to process and platform independent. With all this, XML is a suitable means to represent middle-tier data models, to store and relay data in an information exchange process.

An XML Schema along with XML data files can be used to build a middle-tier data model. The schemas define rules to serialize and deserialize the information from source databases. It can convey information such as table names, column names, relationships among tables, key constraint, data types, etc. The schemas also depict the mapping between the data models.

There are many ways to implement data transfer between systems. There are tools specially designed and built for a specific system, limited to specific data schemas. Other tools are more generic, capable of being configured to carry out transferring tasks among a variety of systems, even different application domains. The principle of steps for transferring data are extracting data from the source, and then importing data into the destination. In the cases where the schemas are identical, the process is straightforward. In reality, the problem is complicated when dealing with heterogeneous schemas from different systems. Finding a matching mechanism between the systems is the challenge. To be able to change, merge, and coalesce the data to match each side. This is a problem that requires a good understanding of both systems and all schemas involved, as well as a tool for modeling and finding their logical relationship and semantic connection inside the data.

When creating a tool, with the goal to create such a mechanism between the systems, there are some requirements. The tool must have a module to access and configure the parameters related to the data transferring process to change data source, data format or type, and constraints. It also needs to be possible to decide which tables or relations are involved in the source database, how to organize and tailor it to match the schema of the received system.

Having the export and import as two separate steps while using an intermediate data model as a middle data storage in the transferring process is also possible. The model not only stores but also makes the schema transformation. The advantage with this solution is that the procedures are more separated, having importing and exporting as two separate steps in the transferring process. This brings flexibility, makes schema transformation smoother, makes the whole process easy to manage, and makes it possible to reuse and share parts of the solution.
2.3 Possibilities of heterogeneous database types

An approach to create a system that can communicate with different type of databases was found in the answer to a question at [http://softwareengineering.stackexchange.com](http://softwareengineering.stackexchange.com) How are abstract database interfaces written to support multiple database types? David Packer gave an answer, Decoupling an application using abstraction. It felt very fitting for the prototype. He discusses and shows a short guide on how to create a program that at run time could choose to use a specific implementation of an interface.

The answer is to have a database interface, and two specific classes, one for query representation and one for representing the result of a query. Every time the program needs to adapt to a new database type, it is just a matter of making a new implementation of the interface and tell the program to use this new implementation in some configuration setting, for example, a switch-case solution. The methods in the interface are responsible for, beginning, rollbacking, committing, and checking if the program is in a transaction. By then having the program use the interface methods there is nothing in the code that has to be changed when switching the program to communicate with another type of database, it is a simple matter of configuration.

2.4 JSON and XML

Both JSON and XML are well-known ways to represent data in a human readable form, during data transfer between different applications. We felt the need to learn more about their differences before making a decision of which to use. A comparison table with the differences from an objective view was used for this, "Difference between JSON and XML" by Poonam Dhanvani. To complement Table 2.1, two articles that are biased towards each of the approaches were chosen as well, "Working with JSON in C#" by Jennifer Marsh, and "JSON vs. XML from a .NET developer’s perspective" by Denzel D.

XML originates from Standard Generalized Markup Language (SGML), and is more structured than JSON, making it easier to validate the data. JSON, Although derived from JavaScript, is a flexible format that can be used with many programming languages, it is a bit easier to read as a human, compared to XML. In table 2.1 below there is a short comparison of the two.
According to the article by Jennifer Marsh, there are three key methods when working with JSON in C#:

- Use a library that can handle most of the work for you.
- Work with DataContracts that are built-in to the .NET framework.
- Write your own parser to convert JSON in strings to a suitable collection of objects.

Using libraries simplifies the process of working with flexible data structures in JSON format. Json.Net and Json-sharp are two of the most popular libraries. The principles when using libraries is to, add the library to the project, then use a deserialization method from an appropriate converter class, pass JSON data as a string to the converter, and then finally receive the result as a .NET framework object (often a collection of some kind).

From .Net Framework v3.5 and onwards, a JSON converter is included in the framework. Unlike the libraries, using these classes requires more knowledge of the web service that is to be worked with. Due to this, it is a slightly more advanced technique. To use this converter, a reference to System.Runtime.Serialization is needed in the project. It is also a necessity to define the types of objects as DataContract attributes.

The possibility to write code that converts JSON as strings to a format that is fitting for the application exists. However, this is usually an extremely difficult process, as it shares many similarities with compiler design.

### 2.5 CORBA

The content of this section is based on the article "The Rise and Fall of CORBA" by Michi Henning. He discusses CORBA, which is interesting for this thesis because it is one of the first solutions for heterogeneous distributed system communication.
This material is included to give an insight of what has changed since CORBA was introduced, to gain knowledge of which paths to avoid.

CORBA was created in the early ’90s. It gives developers a tool that allows them to build heterogeneous distributed applications with relative ease. However, following the large and complex specification is hard, and much of it has not even been implemented as a proof of concept. Some sections are essentially unimplementable, and in the cases that they are, portability is not provided. Threaded applications are inherently non-portable since the specification ignores threads. Adding to this, CORBA has no language mapping for C# or visual basic, completely ignoring .NET framework.

*Writing any nontrivial CORBA application was surprisingly difficult. Many of the APIs were complex, inconsistent, and downright arcane, forcing the developer to take care of a lot of detail.*

- Michi Henning.

When communicating, CORBA does not utilize a clear data format, such as XML or JSON. Which in CORBA’s defense, did not exist at the time it was created. Another problem with the data traffic when using CORBA is that it is not encrypted, and thus a bigger subject to eavesdropping and man-in-the-middle attacks. It also requires a port to be opened in the corporate firewall for each service, which conflicts with the reality of corporate security policies.

### 2.6 SOAP and REST

This section is based on the article "Understanding SOAP and REST basics and differences" by John Mueller. The article discusses SOAP and REST, two ways for distributed systems communication, both newer than CORBA. The information gained from this investigation will be used to chose if the prototype’s communication should be based on SOAP or REST.

SOAP and REST are ways of accessing web services using the HTTP protocol. They both rely on well-established rules that everyone has agreed to abide, in the interest of exchanging information. REST as an architecture style does not require processing and is naturally more flexible. SOAP has a more rigid set of messaging patterns compared to REST.

SOAP exclusively uses XML as data representation. The XML in requests and responses can become extremely complex, thus cumbersome to use. In some programming languages, these messages have to be built manually, which is problematic due to SOAP’s intolerance to errors. However, some languages can use shortcuts provided by SOAP. The shortcuts helps to reduce the effort of creating requests and parsing responses. The programmer may never even see the XML. An important SOAP feature, is the built-in error handling. If a problem occurs during a request, the response will contain error information as to what the reason was. If the programmer only have access to the client and not the server, this is a really good feature.
REST provides a lightweight alternative to SOAP. It is not limited to XML to provide responses. CSV, JSON, and RSS are common options instead. The output can be obtained in a form that is easy to parse with the language used in the application. There are four different HTTP verbs that REST utilizes to perform tasks. GET, POST, PUT, and DELETE.

Comparing the two against each other, both have advantages over the other.

- SOAP is, language, platform, and transport independent, while REST requires the use of HTTP. SOAP works better in distributed enterprise environments, due to RESTs assumption of point-to-point communication. It is also more standardized and provides built-in error handling.

- REST is easier to use and is more flexible. It does not require expensive tools to interact with web services. REST has a smaller learning-curve than SOAP and is closer to other web technologies in design philosophy. It can be more efficient due to being able to use smaller message formats than XML and is faster since it does not require extensive processing.

### 2.7 Tunneling SQL through HTTP

This section is taken from the article "SQL tunneling through HTTP" by Thomas Kvamme and Shakil Ahmed. They discuss and develop a system with the possibilities of using an HTTP tunnel to access a database without having to provide access to the database native communication port. This is included in the investigation since it gives insight on how one can send database commands through HTTP.

A tunneling system to access a database without providing access to the database native communication port can be created. This is useful if a database needs to be available on insecure networks or if firewalls between client and server does not accept anything other than the normal HTTP communications. A client usually communicates with a database server using a native communication port, where access is controlled by the firewall between. However, an SQL-tunnel can successfully be developed, supporting connectivity from both web browsers and applications. The system tunnels the traffic across HTTP. The client can therefore communicate with the database without access to the database native communication port. In some cases, creating such a tunnel can even outperform the native communication protocol provided by the database vendor.
3 Analysis of investigation

In this section, the investigation is analyzed. The material is evaluated on the basis of needs for the prototype to create the most suitable implementation of the prototype. An important note for this section is that the prototype is developed in the .NET environment, during a rather short time-frame. Thus, simplicity and implementation speed is important, as we do not want to spend time "reinventing the wheel". Since the prototype will deal with heterogeneous data and databases, flexibility and genericness are also two important criteria.

The analysis is divided into three parts. **Transferring and representing the data**, **Database synchronization and change tracking**, and **heterogeneous databases**. Each section refers to the material presented in the investigation section. However, the important points from each section that affected the design decisions are summarized within the analysis.

3.1 Transferring and representing the data

We chose to first focus on how to represent the data while transferring it. This part is isolated from both retrieving and submitting the data from different databases. No matter what the later decisions are, this decision will not have to be changed. The only way in which we would have to change data representation is if we chose to use a transport protocol that does not support the data representation. Since future versions of the prototype could end up in a cloud environment, the transport protocol needed to be one that is commonly used in the web today. Using this standpoint we ended up with questions about JSON, XML, REST, SOAP, HTTP and distributed system communication. The material behind this discussion can be found in sections 2.2, 2.4, 2.5, 2.6, and 2.7.

Based on the findings in section 2.4, we felt that using JSON would be a more fitting approach for the prototype. This is because the system is in control of both sides of the transfer. Thus, the extra structure and validation possibilities that XML brings does not outweigh the lighter syntax and direct array support that JSON offers. JSON also seemed faster and more convenient to implement and use. We also learned that although JSON is not developed for .NET, it is still highly usable in the framework. There are even different approaches to choose from. Thus, choosing JSON over XML will not limit us in any way. We first wanted to write our own parser to have full control of how the serialization work. That decision brought too many problems and forced the project into a direction that was not part of the thesis. Instead, we chose to use Newtonsoft's JSON-converter[^1] This resulted in the possibility to spend more time focusing on other areas.

We had to evaluate the protocols available for the communication/transfer of the data after making the decision to use JSON. This brings us to section 2.5. At first, CORBA sounded very appealing to use. However, after learning how troublesome and complex it was, we understood that this would be a very unnecessary path to follow. Instead, the option was to use REST or SOAP. With the information from section 2.6 we chose to go with a REST-like approach. The drawbacks that

[^1]: http://www.newtonsoft.com/json
REST has to SOAP is that it requires the use of HTTP, assumes point-to-point communication, and does not have the same error handling possibilities. Since we aimed to be able to use this in a cloud environment, the HTTP requirement is not a problem. Neither is the assumption of point-to-point communication since this is exactly what the prototype will do. Lastly, because we are in control of both ends of the communication, losing the error handling did not feel as problematic. The benefits of a REST-like approach is that we could stick with the choice to use JSON instead of XML. Rest is also said to have a smaller learning curve, making the implementation less complicated.

3.2 Database synchronization and change tracking

The prototype works as a database synchronizer. The only difference is that the prototype does not need to live synchronize. It also must be able to fetch data from a number of different database types. Although, the target database is always a SQL database. Section 2.1 and section 2.2 have solutions to synchronize databases that are heterogeneous in different ways. In section 2.2 a system that utilizes an XML schema to map the corresponding relations between two different databases is utilized. This schema makes it effortless to transfer data between the two databases without having to remodel the data manually every time. This sounds like a good solution, but it is not applicable in our solution. We are going from one database with a representation, to a database without any representation at all. Thus, it is simpler to stick with the same representation in the target database. In section 2.1 a system that utilizes look-up tables to adapt SQL-queries is described. It also logs all queries that did not adapt accordingly in a specific file. Thus, a manual conversion could be made in those cases.

We chose not to follow any of the above described approaches. Instead, we use configuration tables in the database that we want to gather data from. These tables are similar to the look-up tables, but it has to be stated exactly how the data from a specific column is to be represented in the target database. Most current settings for configuration in Trimmas systems uses the same method. The choice was made to also use this approach in the prototype to further follow their ways. This only solves the mapping problem of the synchronization, tracking changes and avoiding transferring data that will create doublets in the target database is still a problem. However, the prototype still operates as planned if the target database is emptied before each data transfer. Therefore we made the decision to postpone dealing with this problem.

3.3 Heterogeneous databases

The hardest topic to find previous works within is the possibility to create a system that does not care what type of database it communicates with, no matter if it is a SQL, no-SQL, flat-file, or any other database type. This means that not only the content of the database is heterogeneous, but also the database type. It is more of an implementation specific area, not a scientific, and that is probably the reason why it was hard to find information. However, we found an answer that felt fitting for the prototype, and it is described in section 2.3.
With the use of abstraction it is possible to decouple the application from a specific database type implementation. It is therefore possible to configure the prototype to work with different database types, without the need if changing any code. This also means that a new implementation would need to be created for each new database type that the prototype support. This is a good approach since it is easier to understand a small interface compared to the code of the whole prototype.
4 Resulting prototype

In this section, the resulting prototype is explained. The decisions made in the analysis are summarized and some overview pictures of the system are shown.

The resulting prototype which implementation is described in section 5, is a distributed system divided into three programs. Two of them can be seen as clients and communicates with the source system database. The last program communicates with the target database and works as a server for the client programs. One client is responsible for fetching and transferring the structure of the source system database to the server program. This enables the server to set up the corresponding tables in the target database. The other client is responsible for fetching and transferring the actual data from the source system database, allowing the server program to insert the data into the target database. Two configuration tables are used, `transfer.Tables` and `transfer.Columns`. These keep track of what to transfer and are further described in Appendix B.

The system communicates with a REST-like transfer protocol using HTTP and JSON-strings. The target database is always an SQL database while the source

![Figure 4.1: Simplified flowchart of the prototype](image)

The flow of the prototype is illustrated in Figure 4.1. It shows how the clients communicate with their database, in what state of the process the transfer occurs and how the server program operates. From the flowchart, we can interpret that the setup client has to make the server create the tables before the update client can make it fill the tables.

The system communicates with a REST-like transfer protocol using HTTP and JSON-strings. The target database is always an SQL database while the source
system database can differ from different source systems. The correct database type must be configured before running the program. However, there are no other steps needed to configure the system to work with other databases. Creating support for a new database should be easy and straightforward for someone with knowledge about the source system’s database and C#. How to do add support for a new database and configure the database type is described in section 5.4. Since the prototype will be used from different source systems at the same time, the current solution is to have one instance of the server program for each source system. This means that any amount of clients can be connected to one server. However, each source system needs a separate server. Why the prototype is implemented in this manner is covered in section 7.2.

The system as a whole, and each subsystems outgoing communication and task-space are visualized in Figure 4.2. The green area (right side) corresponds to the Server-program, while the blue area (left side) corresponds to the Client-programs. In the blue area, two database-types numbered 2 and X are visualized to show that it is possible to implement the interface for any database type. More specific information about methods and classes can be found in section 5.

Figure 4.2: Overview of prototype
5 Implementation

This section explains the implementation of the prototype. The prototype is made in Visual Studio using C#. The prototype needed to support genericness in the case of database type. However, it is only implemented with one type. The database language in the implementation is SQL since it is the most frequently used database in the source systems at Trimma. There is more information about the design decisions in section 3.

The system is divided into three areas, collecting, transferring, and storing the data. The implementation of the prototype is also done in this manner. Thus, we started by writing three smaller programs to test the decided approach for each area.

- **The collecting program** was a simple console application that fetched data from an "unknown" database, then outputted the result with a nice layout in the console.

- **The transferring program** created a table-representation class, serialized it to a JSON-string and then deserialized it to the table class again. When this program was done it was expanded to also send the JSON-string through HTTP before the deserialization.

- **The storing program** contained code to convert the same table representation class to SQL insert queries and perform them on a database.

When all three programs worked, we created a new solution and used the knowledge gained from writing the three smaller programs to create the prototype. The functionality for each area in the prototype is described in the following sections.

5.1 Collecting the data

In the prototypes source system side, heterogeneity in database type and data are two big problems. At the time of coding the prototype, both the type and content of the database is unknown. To deal with this, the prototype is built according to the decisions in analysis sections 3.2 and 3.3. This means that heterogeneous database types is dealt with by decoupling the prototype using abstraction. Therefore an interface for database communicators is in the prototype. The interface is implemented for each new database type that is supported, the prototype then choose the correct implementation at run-time. More information about the interface is found in Appendix A.

The data heterogeneity is solved with the use of the configuration tables described in Appendix B. From these tables we can fetch information of how the data should be represented in the target database. Information about the database to connect to is configured in a specific file, described in Appendix E. If the prototype is configured to work with SQL-Server databases there is a configuration for the setup program. If enabled, the program will automatically fill the columns configuration table based on the information in the tables configuration table.
5.2 Transferring the data

The prototype uses a REST-like approach as a transfer protocol together with a JSON data representation during the transfer, as stated in section 5.2.1. How the data is serialized before and deserialized after the transfer is described in Appendix C.

The HTTP transfer itself is set up with an HttpClient from the sending side. It is used to send HTTP requests to a given server. On the receiving side a HttpSelfHostServer is used as a server, to receive HTTP requests. The server is used together with an implementation of the abstract class ApiController. It is used to map specific requests to the correct method of the server program. More technical information, and a description of the API is located in Appendix D.

5.3 Storing the data

The request content which the server receives can be very large. The program uses a method called bulk insert to insert large amount of data into the target database. Exactly how the bulk insert works is described in Appendix F. The server uses the same method as the clients to configure the connection to the target database, this is described in Appendix E.

5.4 Implementing support for another database type

In order to add another database type (e.g., OracleDB) to the solution all that is needed to be done is to implement the interface IDatabase. The interface has three methods, Connect, Disconnect, and QuerySelect. The interface is further explained in Appendix A. QuerySelect is the hardest one to implement and will deviate from the other interface implementations the most. It might be rewarding to check the code for the SQL implementation before creating a new implementation of the interface. It may contain useful tips and methods of implementation towards the interface that are not database specific.

In order to use the newly implemented interface three steps needs to be performed.

1. Add the implementation to the DatabaseImplementations folder in the project files.
2. Add a value representing the implementation to the enum DatabaseType in the DatabaseSettings class.
3. Change the value of the key database_type located in the App.config file in the project folder to the newly added DatabaseType.

---

There are some extra features that the current SQL-Server implementation has which are not demanded by other implementations. These include being able to set up the columns configuration table automatically based on the tables configuration table. In cases where the implementation will be used by only one source system, it might be easier to manually fill the second configuration table instead of implementing this feature. In order to implement this feature, some manual changes to the actual code in the main method of the setup client has to be made.
6 Prototype evaluation

In this section, an evaluation of the prototype is given. The prototype is tested and fully usable in its current state. It can transfer data from one database to another database which has no knowledge of how the data is represented. All that is needed is the connection information to both databases and the name of the tables that should be transferred. With this outcome, we believe that the result of this project is successful.

Nonetheless, by monitoring the prototype in a test environment together with Trimma we realized that there are some necessary additions. Some of them would result in great improvements, others are a necessity before the system can be used practically in a live environment. These additions and the reasons behind them are presented in the list below. They are also individually discussed in section 7.

1. Security
   - The data is not encrypted during transfers. In cases of sensitive data, encryption is a necessity.
   - There is no authentication needed to insert data into a database. Currently, the client can be used to insert false data into the target database. All that is needed is the address to the server program’s controller method and the table structure.

2. Change tracking
   - If the target database is not cleared before transfers it will contain very large sizes of data that already exists in the target database. This is due to the inability to detect data that is already represented in the target database. A large amount of data is also transferred in vain.

3. Threaded server program
   - One instance of the server program is needed for each source system since the server program only supports one database in the configuration. In addition, the server does not utilize threads to receive requests.

4. Insert optimization
   - Bulk insert is slightly less efficient than single insertion on small data sets.

5. Graphical configuration interfaces
   - Having to open configuration files and change settings is not a preferred way to configure a program.
7 Discussion

In this section, a discussion is held about how and why the resulting prototype changed as it did. Further development possibilities are also discussed.

During the project, new suggestions and facts where brought to us from Trimma. These gave an opportunity to change some goals that we had aimed for previously.

1. At first we thought we needed to manipulate the data and reach the DW (DataWarehouse) stage, where the data has been modified to be used by INSIKT. Rather early we found out that we only needed to go to the SA (StagingArea) stage, where the data is a hard-copy of the data from the source system. This was a necessary change since it would have been impossible to implement both steps and to keep the demand on heterogeneity. However, it did not affect much since we had not gotten that far with planning when this was brought forward.

2. We only needed to implement support for SQL-server but prove that the solution was generic enough to easily implement support for other databases. The original plan was to implement support for two or three databases. Therefore this change allowed more time to increase the quality of the prototype.

3. We should use bulk insert to fill the target database. The plan was to only use single insertions, and the effects and benefits of this change are discussed in section 7.2.4.

4. We did not have to implement any kind of security. However, in Section 7.2.1 we discuss possible additions.

7.1 Deviations from the project plan

Some points in the list above made us deviate from the project plan. These changes affected the result and made it deviate from the project plan. Nevertheless, the main deviation was that we had hoped to find more concrete solutions for each problem area stated in the project plan. The areas were, collecting, transferring, and storing the data. However, there were not enough previous work made in this manner. Thus, the investigation was slightly less structured making it impossible to do the comparisons that were originally planned. Instead, the analysis was made more fluent and qualitative instead of quantitative, with a more clear comparison. This made the analysis difficult to present in a statistical way. In the end, we think this path led to a resulting prototype with the same quality as the original plan would have created.
7.2 Further improvements

Based on the prototype analysis in section 6, we find it important to discuss some improvements that should be implemented before running the prototype in a live environment. The improvement that we find mandatory is adding security. The second most important upgrade is to implement some way of change tracking. If the prototype is to be used with many source systems simultaneous, it would be a great improvement to make the server program threaded. Nonmandatory improvements that would still bring a lot to the table include adding a graphical configuration interface, and optimization for inserting the data into the target database.

7.2.1 Security

We had no ambition of adding security to the prototype due to time constraints. However, encrypting the data is crucial in order to use the prototype in live environments. Together with Trimma we also have the belief that in besides encryption, some authentication that ensures that the transferred data has access to the target database is necessary. Trimma currently has projects where both these security issues are covered, and we believe it should not be hard, or time consuming to integrate that security into the prototype.

7.2.2 Threaded server program

The current solution forces each source system to have one server program. That means if this is to be used, one server program has to be started for each source system before the transfer can start. The cause for this is that the settings in the server program have to be manually configured to the correct database. If there were some kind of authorization to map requests to the correct database, the server program could instead be developed into a threaded program. A threaded program could deal with requests from several source systems at the same time. This would be an improvement compared to the current need for several instances of the server program.
7.2.3 Change tracking

We had hoped to add an implementation of change tracking in the prototype. Due to time constraints, this was not possible. However, we have at least found some material, and have ideas on how this issue could be approached.

In SQL-server 2016 there is support for tracking database changes and easily mirror a database. However, this approach is not applicable to the prototype. The reason is that it is not possible to enforce the source system databases to be of this specific type, nor to ensure that the owner of the source system would grant us this information. Another approach that is used by the system explained in section 2.1 is to capture all queries executed to the source systems database. This is not possible since we are not in power to change the source system. The section also names that WisdomForce and Oracle have solutions for database synchronization. However, they have their respective limitations. WisdomForce only works for data and not schema synchronization while Oracle's solution only works with Oracle databases.

In a discussion with Trimma, a possible solution was brought forward. It was based on adding an extra column to the larger tables in the source database. The column would contain a date and time of the last update for each row. With this solution, it would be possible to only transfer "new" rows. This would solve the problem of not sending duplicates, as we would only send rows that had a newer date than the last transfer. However, if it is an old row that has changed, we would still have the problem of having two "copies" of the same row in the target database, only with different values. One might think that this problem could be solved with the use of primary keys in the database. Yet, almost none of the source systems databases has primary keys and adding this feature now is not an option.

7.2.4 Insert optimization

At first, we had the idea to have one insert query for each row during the update phase. This idea was implemented so far as to only execute the queries. At this point, a meeting about the progress of the project was held with Trimma. They asked that we would investigate bulk insertion, as the amount of data could become too large to insert one at a time in a good manner. This decision added some complexity to our solution as the current table representation class could not be used with the available bulk insert method. We had to set up a DataTable with the exact same structure as the database table on the receiving side instead of having a method that created simple strings where each value was easily mapped to the correct column. This forced us to communicate more with the database to set up the DataTable accordingly. In cases were the amount of data to insert is small bulk insert might be slower than single insertions. In other cases, it will be significantly faster. Anyway, since this happens on the retrieving side of the transfer, the database is known. Working with a known database made the implementation straightforward. We were able to fetch the table structure from

https://docs.microsoft.com/en-us/sql/relational-databases/track-changes/
track-data-changes-sql-server
the INFORMATION_SCHEMA.COLUMNS table that exists in the SQL-Server databases.

It might be interesting to use both of the approaches in future versions of the prototype. By combining them to be used at a data size where they would work better than the other alternative the prototype would be more effective.

7.2.5 Graphical configuration interface

There are two files that need to be edited to configure the prototype. One file for the source system side and one for the target database side. We have used the AppSettings tag in the App.config file to store and read these configuration values. The App.config file is located in the project folder and is used to configure variables without changing any code. It would not be that hard to create a graphical interface to access and change these values. By doing this change, the system would be much more usable for someone that has not been involved in the development of the prototype. It would also be easier to validate the value of said settings.
8 Conclusion

It is possible to create software that can be configured to fetch data from any given database with the use of standard frameworks and protocols. It is also possible to convert the data to fit into a database of choice, yet keeping the same structure.

It is important to keep the software open to easy changes since the type of the source database can vary. With this in mind, we evaluated that the best way is to decouple the application with abstraction to make it easy to implement unsupported database types. A REST-like protocol that transfers JSON-string through HTTP is a great way of transporting the data since all three are already widely used and well known. This makes it easy to further develop the prototype. Also, utilizing the software in cloud environments will be possible.
References


Appendices

A Database interface

The database interface is named IDatabase and contains three methods:

- `bool Connect();` - Connects to the database. Return true if successful. Assures that the QuerySelect can be used freely while connection has not been disconnected.
- `void Disconnect();` - Disconnects from the database.
- `Table QuerySelect(DbQuery query);` - Performs a query corresponding to an SQL SELECT with the information in the DbQuery. Returns a Table with the result from the query.

In the method list above, `QuerySelect` is depending on two classes that represents the query and result for a database query. `DbQuery`, and `Table`.

`DbQuery` has three fields.

- `string tableName` - Name of the table.
- `string tableSchema` - Name of the table’s schema.
- `string[] columns` - Array of the table’s column names to be fetched.

`Table` has four fields.

- `string name` - Name of the table.
- `string schema` - Name of the table’s schema.
- `Dictionary<string, string> columnTypes` - Dictionary containing `<column name, column type>` relation. The column type is the corresponding C# type.
- `List<Dictionary<string, string>> rows` - A list with dictionaries containing `<column name, column value>` relations.

`SQLDatabase` is the implementation of the `IDatabase` interface, it has the following algorithm in the `QuerySelect` method:

1. Create a query string based on DbQuery:
   - `SELECT columns FROM tableSchema.tableName;`
2. Perform the query on the database.
3. Parse the result to a Table.
4. Return the Table.
The system is depending on the existence of one schema and two tables in the source system database. A schema named `transfer` and the two tables, `Tables`, and `Columns`, these are referred to as configuration tables. Of course, their names can be changed in the App.config file of the project’s client folder. The relation between the tables is shown in Figure B.1. One configuration table with the names and schemas of all the tables that contain columns that are to be transferred. These tables are given an ID that is a primary key, and thus, no two tables can have the same ID. In the other configuration table, containing info about the columns that are to be transferred, the same ID is used as a foreign key to connect each column of the tables to the correct table.

Each table in the target database will be created with the exact same schema name, table name and column settings as the source table.

![Figure B.1: Configuration tables.](image)

The SQL queries used to create the two configuration tables are:

1. CREATE TABLE transfer.Tables (  
   table_name NVARCHAR(128) NOT NULL,  
   table_schema NVARCHAR(128) NOT NULL,  
   table_ID int NOT NULL IDENTITY(1,1),  
   PRIMARY KEY(table_ID)  
   );

2. CREATE TABLE transfer.Columns (  
   column_name NVARCHAR(128) NOT NULL,  
   column_type NVARCHAR(128) NOT NULL,  
   column_nullable VARCHAR(3) NOT NULL,  
   table_ID int NOT NULL,  
   FOREIGN KEY(table_ID) REFERENCES transfer.Tables(table_ID),  
   UNIQUE (column_name,table_ID)  
   );
C JSON serialization

The queries to the source system databases return their result in the form of the `Table` class described in Appendix A. This class is also used in the purpose of serializing and deserializing the JSON-strings that is the content of the transfer request. To perform the conversion between table class and JSON-string the system uses the class `JsonConvert` from the `Newtonsoft.Json` package. When serializing, it is a simple matter of using the method `JsonConvert.SerializeObject(List<Table>)` to create a JSON-string that describes all tables. For deserialization, `JsonConvert.DeserializeObject<Table[]>(jsonString);` returns an array of tables.
The receiving server uses a controller approach, with one controller that has two methods. One for the update and one for setup. They both only answer to HTTP POST request methods, and the sender needs to use their exact names to send requests. The message, or in this case, a JSON-string, is packaged into the content of the request. The sending side uses a dictionary together with an enum variable called action to keep track of what is possible to do on the receiving side. The dictionary maps the action to the correct server address. It will only be the given actions to choose between, rather than having to remember and write the whole address as a string. The string achieved from the dictionary is appended to the base address of the server before sending the request. In Table D.1 below the prototype’s API is presented, it describes the address that is added to the base address and which combination of HTTP method and action that is connected to it.

<table>
<thead>
<tr>
<th>Method</th>
<th>HTTP method</th>
<th>enum action</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>POST</td>
<td>update(0)</td>
<td>api/transfer/Update</td>
</tr>
<tr>
<td>Setup</td>
<td>POST</td>
<td>setup(1)</td>
<td>api/transfer/Setup</td>
</tr>
</tbody>
</table>

Table D.1: Transfer API of the prototype.
E  Configure connection to target database

To set up a connection to a database, the program needs to create a connection string that contains information about what user and password to use on what data source and database. The program fetches these four arguments from the App.config file that is located in the project folder, and then builds the string based on them. In the appSettings section located in the App.config file, there are four keys:

- **data_source** - the data source that the database is in.
- **initial_catalog** - the database name.
- **user_id** - the user id that wants to connect to the database.
- **password** - the password for the user id.

This approach makes it easy to configure each part of the prototype to work with different databases. It also enables people that are not familiar with the code of the prototype to configure it.
F  Bulk insert

The prototype uses the bulk insert method \texttt{WriteToServer(DataTable)} from the \texttt{SqlBulkCopy} class which exists in the .Net framework. To create the \texttt{DataTable} class representation of the \texttt{Table} class described in appendix A, the program first fetches the table’s layout in the target database so the \texttt{DataTable} can be created with the same structure. After that, each row in the \texttt{Table} class is modified to a \texttt{DataRow} which is then appended to the \texttt{DataTable}.


\footnote{https://msdn.microsoft.com/en-us/library/system.data.datarow(v=vs.110).aspx}