Migrating a Database from Git to a Relational Database

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor’s thesis

Degree programme in

Computer Applications

Spring, 2023

Eric Telkkälä

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus / Computer Applications — Tiivistelmä

Tekijä	Eric Telkkälä	Vuosi 2023
Työn nimi	Migrating a Database from Git to a Relational Database	
Ohjaaja	Tommi Lahti	

Tiedon helppo saavutettavuus ja manipulointi on tärkeätä, kun sen kanssa työskentelee. Hidas tai monimutkainen pääsy tietojärjestelmään hidastaa työntekoa ja tehokkuutta. Opinnäytetyön tilaaja, Fail-Safe IT Solutions Oy, tilasi työkalun, joka siirtäisi tietoa Git-järjestelmästä tietokantaan. Opinnäytetyö tutki syitä miksi kyseinen siirtyminen oli pakollista, työkalun rakentamista, mahdollisten vanhojen versioiden säilyttämistä sekä lopullisen tietokuorman koon pienentämistä relaatioiden avulla.

Tutkimusosio tutki Git-versiohallintajärjestelmän epäsopivuutta tietokantaratkaisuna siitä puuttuvien olennaisten tietokantatoimintojen osalta, syitä Java 11-ohjelmointikielen käyttöön tässä työkalussa, perustietoa Git-versionhallintatyökalusta, JavaScript Object Notation'sta (JSON), sekä kryptografisista tiivisteistä ja niiden luonnista. Viimeisenä tämä kappale käsitteli ohjelman testausta sekä sen tuomista hyödyistä ohjelmoinnissa.

Käytännön osuudessa työkalun ohjelmointia käsiteltiin askel askeleelta ja paikoitellen esimerkkejä ohjelmasta annettiin pseudokoodina, sillä tämä opinnäytetyö sisältää salassa pidettävää materiaalia, joten kaikki ohjelmistokoodi ja data on epäaitoa. Versioiden säilyttäminen oli mahdollista, kuin myös lopullisen kuorman pienentäminen relaatioilla, mutta tämä vaati ohjelmointirajapinnan (API) muuttamista, jotta se osaa käsitellä mahdollisia relaatioita.

Opinnäytetyön aihe oli tärkeä osa asiakkaan yritystoimintaa ja opinnäytetyö mahdollisti ohjelmistorajapinnalla hallittavan järjestelmän käyttöönoton. Opinnäytetyö auttoi asiakasta tekemään valintoja erilaisten lähestymistapojen hyötyjen ja haittojen perusteella. Tämä opinnäytetyö ei kuitenkaan kuvasta asiakkaan oikeaa ympäristöä, sillä se on osa asiakkaan immateriaalioikeuksia.

Avainsanat	Git, versionhallinta, tiedon kaavinta
Sivut	28 sivua ja liitettä 5 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology Abstract
Author Eric Telkkälä Year 2023
Subject Migrating a Database from Git to a Relational Database
Supervisors Tommi Lahti

The accessibility of data is important when working with it. Slow access times, complicated access, or non-standard solutions for data can hinder the development of software. The commissioner of the thesis, Fail-Safe IT Solutions Oy, wanted to create a tool to migrate data from Git to a database. The thesis explored the reasons for the migration, building the tool itself, the possibility to retain previous versions of the files as well as the possibility to optimize the final payload via relations in it.

The research part of the thesis explored the unsuitability of Git as a database solution due to its lack of basic database functions, the reasoning for using Java 11 to build the tool, as well as some basic information about Git, JavaScript Object Notation (JSON), and cryptographic hashing and hashes. The theoretical benefits of software testing were discussed as the last topic.

In the practical part, the programming of the tool was discussed in step-by-step sections, with occasional pseudo-code due to the confidentiality of the whole project. All the datasets used are also made-up so as not to breach confidentiality. The versioning could be retained if needed, with a pseudo-code example written. The reduction of the payload size was achieved as well with detailed instructions on how to do that.

The subject of this thesis was crucial for the business of the commissioner, and they were able to gain an advantage of an API configurable system due to this. The thesis did help with decision-making on which different approaches have advantages or disadvantages over others. The thesis does not reflect the actual commissioner’s subject matter due to it being the commissioner’s intellectual property.

Keywords Git, version control, data scraping
Pages 28 pages and appendices 5 pages

Contents

Images, tables

Appendices

Glossary

API	Application Programming Interface: a method for two computers to communicate with each other
CM	Configuration Management: A process of creating and maintaining a product’s configuration and other functions continuously during its usage
CSV	Comma-separated values: A comma-delimited text file for recording data
EOL	End of Life: The end of a product's lifecycle, after which it does not receive any updates from its vendor.
Git	A version control system
HTTP	Hypertext Transport Protocol: An application layer protocol, the foundation of communication on the internet
IDE	Integrated development environment: A program used to write software.
IETF	Internet Engineering Task Force: A standards development organization made up of volunteers
ISO	International Organization for Standardization: A standard development organization
JDK	Java Development Kit: A package of tools used for writing Java applications
JSch	Java Secure Channel: an SSH2 implementation for Java
JSON	JavaScript Object Notation: A file format
Repository (version control)	A storage location for software, that stores metadata alongside the files, which tracks the changes made to the files and file structure
RFC	Request For Comments: A publication for developing and publishing standards, in the form of a memorandum
RSA	Rivest–Shamir–Adleman: A public-key cryptosystem used for secure data transfer, used in older SSH keys for encryption and decryption

SHA	Secure Hash Algorithm: A family of hash functions (SHA-0, SHA-1, SHA-2, and SHA-3)
SNR	Signal-to-noise ratio: In the context of this thesis, the amount of unnecessary code compared to the necessary one.
SSH	Secure Shell Protocol: A network protocol used for operating a network securely.
TDE	Transparent Data Encryption: A technology to encrypt data at rest, meaning that data is encrypted on storage, but not during transport
TOML	Tom's Obvious Minimal Language: A file format for configuration files
URL	Uniform Resource Locator, also known as a web address
VCS	Version Control System: A system that keeps track of and manages changes made to files
XML	Extensible Markup Language: A file format for data transmission and storage

1 Introduction

Configuration Management (CM) is a complicated area of expertise with possibly endless different parameters that could require configurations. When the services start to grow in complexity, so do the amount and complexity of the configurations. Automating this is necessary since configuring more and more complex services would consume exponentially more working hours if this would be done manually.

There are configuration files that need to be stored in a database, where the CM software can access and retrieve them. Some people suggest using Git for this, but it comes with its own set of hurdles, most notably the fact that Git is not designed to maintain security (Nemeth et al., 2017, p. 897). One of the other options is a database, which can be configured as precisely as needed, but requires that configuration to be written, costing more working hours for the company.

This thesis was commissioned by Fail-Safe IT Solutions Oy and aimed to create a tool, that allowed the commissioner to migrate from a Git-based solution into a database-based one. This tool should pull the files from repositories, look for relations inside the files for more efficient storage, and combine all the files into a single JSON object that can be sent forwards. This creation process aimed to answer the following research questions:

- Why is Git not suitable for storing hierarchical configuration data?

- Is it possible to retain the versioning from Git, based on commits/releases?

- Is there a need for relationality between the data?

2 Preliminaries

Ingestion, or data ingestion, means importing and loading data into a system. This does not forbid data manipulation, therefore data can be modified before loading it to a system, which in the case of the thesis is an application programming interface (API) endpoint.

2.1 Java 11

Java 11 is the second-latest Long-term Support (LTS) release of Java published, at the time of writing this thesis, with it being the most popular version of Java according to a survey made by New Relic in 2022. Java 8, the older supported version of Java, Java 8, lost its popularity from 84 % in 2020 to 46 % that year, due to the end date for Oracle's premier support in March 2022. This is also confirmed by a survey made by Snyk, with 64 % of developers using Java 8 in 2019 and 25% of people using Java 11. (New Relic, 2021; Oracle, n.d.; Vermeer, 2020). Due to the fact Java 8 no longer has premier support and is on its way to End of Life (EOL), Java 11 will be the best option for writing software since it's the most popular release, therefore it has the largest community for finding help. The active support for Java 11 comes to an end on the 30th of September 2023 according to an open-source website "endoflife.date" (endoflife.date, n.d.), but its security updates continue for 3 more years after the active updates stop. It should be explored if the newest LTS version of Java, Java 17, should be used for a more future-proof solution.

2.2 Git

Since the project needs to fetch data from Git, it should be talked about what Git is, and why it is being moved away. Git is a distributed Version Control System (VCS), that tracks the changes made to files. A distributed VSC stores the complete contents of the repository on the client machine, instead of checking out individual files from a centralized server, as a Centralized Version Control System (CVCS) does. The differences can be seen as visualized in Figure 1 and Figure 2. This allows VCSs to not only track changes with the main server but also with other collaborators directly.

Figure 1 Centralized Version Control System (git, n.d.-a)

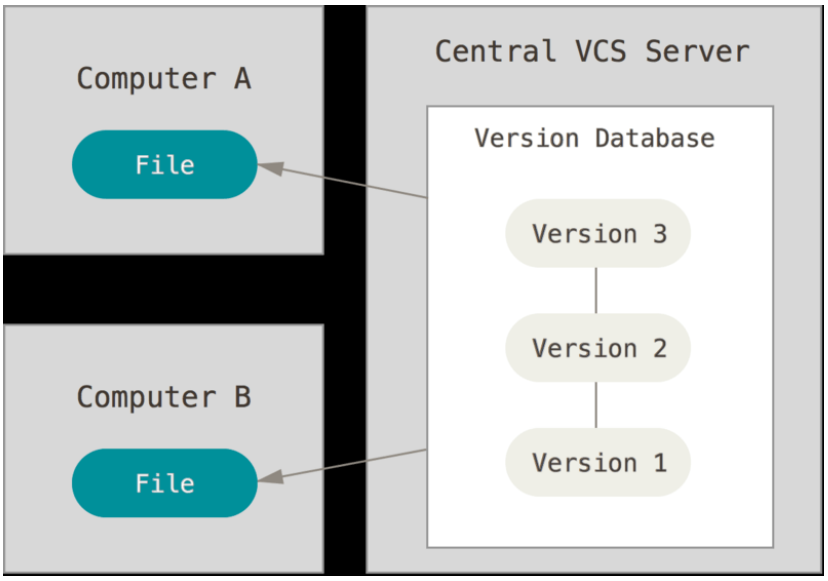
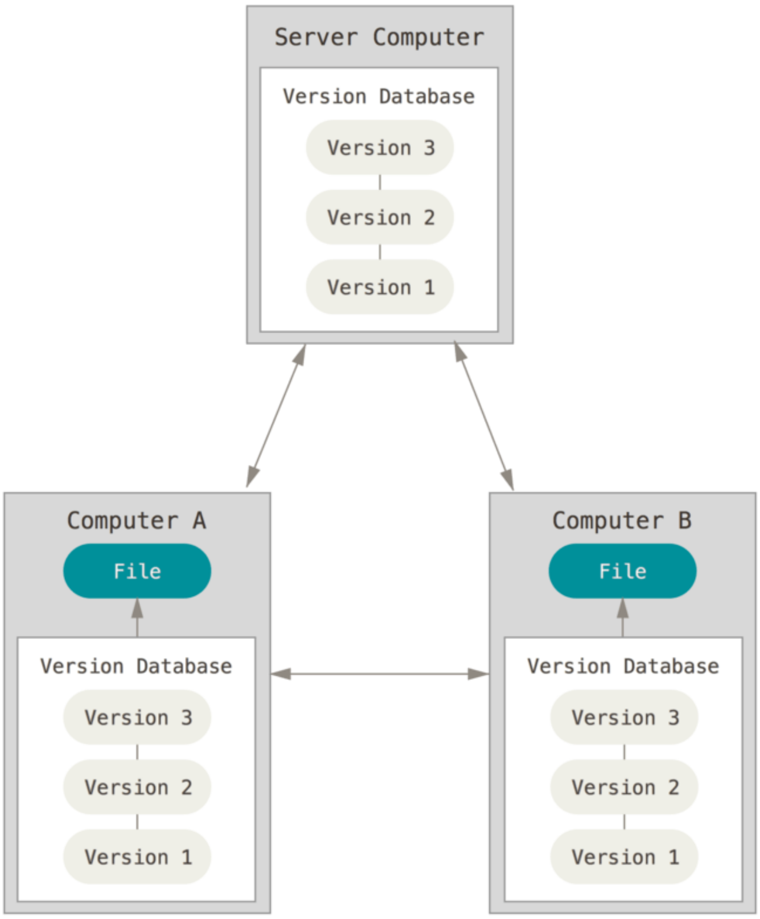


Figure 2 Distributed Version Control System (git, n.d.-a)



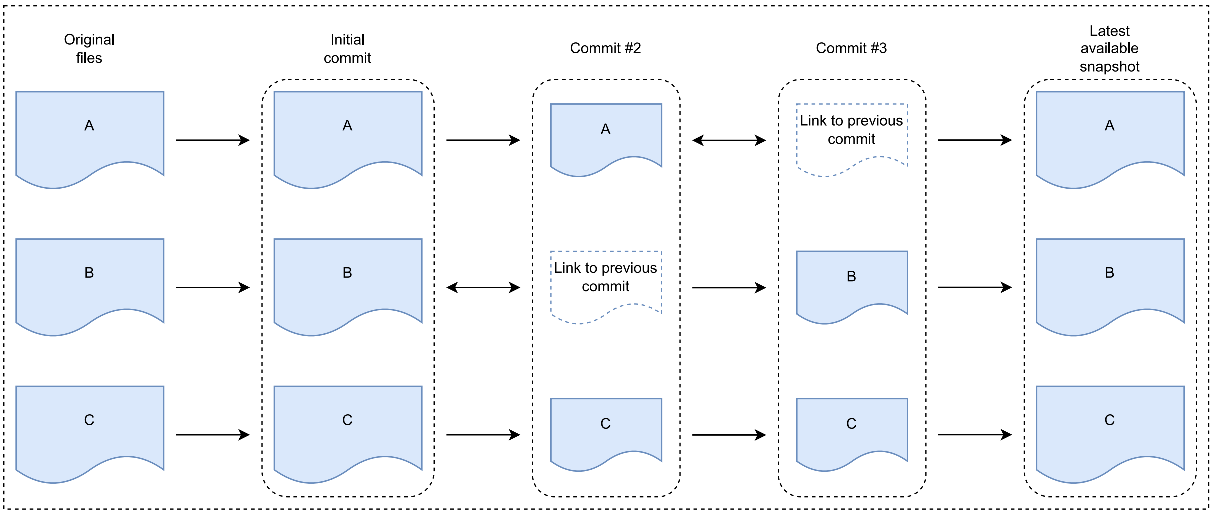
2.2.1 History

Git began its development after the developers of the Linux kernel had a falling out with the source-control management (SCM) system BitKeeper as it removed a free-to-use license from its selection in April 2005 (Chacon & Straub, 2014). This made the Linux kernel community start working on their own VCS. This VCS started self-hosting, meaning that the source of Git itself was hosted on Git, on the 6th of April, and the release of version 1.0 on the 21st of December 2005, 7 months after the conception of the platform (GitHub, n.d.-b; Torvalds, 2007).

2.2.2 Principles

Even though Git is a Version Control System, it should be thought of as managing revisions of files, rather than the version of them (Spinellis, 2012). This type of storage model is called snapshot storage (Laster, 2016). The general idea and execution of this can be found in Figure 3 below.

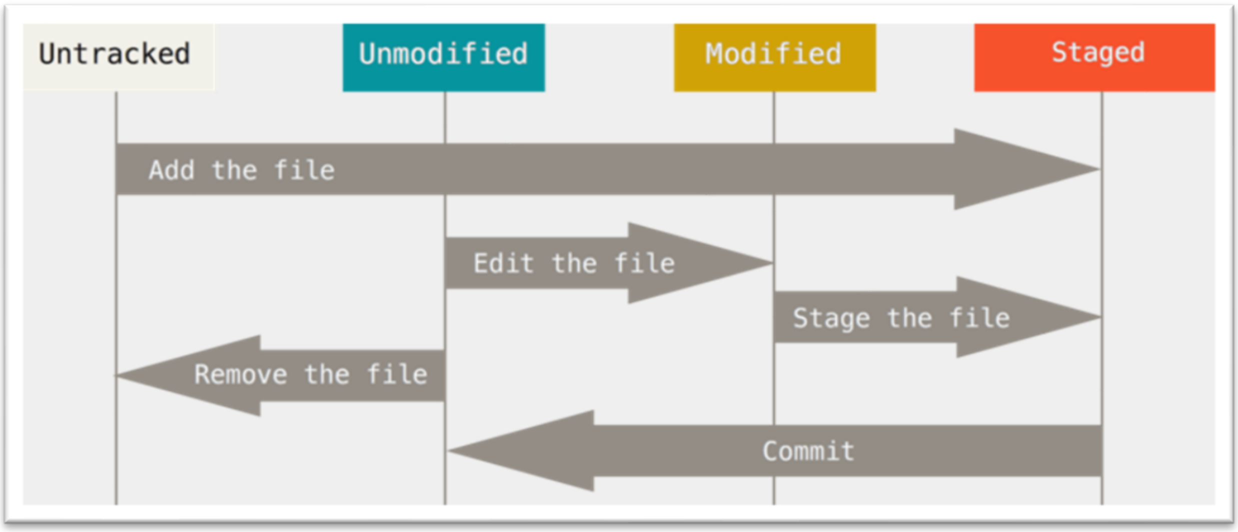
Figure 3 Snapshot storage model



Git operates with pushes and pulls: pushing your locally committed changes sends the changes to the remote server, and pulling in return downloads the changes to your local working directory.

To commit modified files to Git they need to be staged first, otherwise, they will not be sent to the remote server when pushing the changes. This can be done with the command `git add <file>`, for example, to stage a file called README.md, the command would be `git add README.md`. Figure 4 shows that when a file is added, it jumps from untracked to staged, and after committing that file, it moves to an unmodified state, rather than untracked.

Figure 4 The lifecycle of the status of files in Git (git, n.d.-b)



When files are committed, Git creates SHA-1 checksums for the files as well as for the repository tree (the complete layout of the file structure). These checksums are stored in an object database, where each revision gets a new entry to track the changes down the line. This structure of creating new hashes for each revision, and keeping all of them available later, is called a Merkle tree, which Git uses (Shah, 2022). This allows Git to keep track of changes between commits, and as a GitHub user Carl Mäsak (masak) has explained it, the checksums are calculated using the source tree; the parent commit's checksum; the author information; the committer information; the commit message; as well as a message, "commit <length of the information above>" is added to the beginning of the sentence that will generate the checksum. (Mäsak, 2021)

2.2.3 Authenticating the Git user

Since Git is a VCS, it does not implement any security features by itself. Instead, it leaves that to the platform or the server that it is running on to implement it. Nothing stops someone from looking at the code in repositories by default if they know the URL (Uniform Resource Locator) to the repository. As the Pro Git book's protocols chapter (Chacon & Straub, 2014, p. 105 - 110) says: "Git can use four distinct protocols to transfer data: Local, Hypertext Transfer Protocol (HTTP), Secure Shell Protocol (SSH) and Git". The local file protocol retrieves the files from the local storage using hard links and just copies the files in the specified ".git" folder of the file path, for example using the command "git clone /srv/git/project1.git". According to Chacon & Straub, the local file protocol is trickier to set authentication up, due to its behavior as a mountable storage disk. Both the HTTP and SSH protocols operate over the network, and both support some form of authentication with configuration. This configuration is left to the people who implement or host the repository. Lastly, the Git protocol is the hardest to implement and offers no authentication by itself, therefore it disables pushing by this protocol by default but is the fastest because it uses a dedicated port on the server and has a special daemon, meaning that there is a dedicated background process for processing Git requests. (Chacon & Straub, 2014)

According to a blog post by Derrick Stolee (Stolee, 2022), Git shares concepts with typical databases but does not act like one due to it storing Git commit, blob, and tree objects to build the history of a repository. Stolee then mentions in the later blog post that Git is lacking B-trees, which are used to index the data in databases and provide essential operations, searching, deleting, inserting, and sequential search (Comer, 1979, p. 123).

To conclude the chapter, Git is not secure by itself, it requires additional software to manage the rights of the users who can read the repository and its files. Git is not a database solution, but a tool for version control, hence it should only be used for said purpose since Git itself does not offer any tools to search for any data inside repositories, so all the data will always be downloaded regardless of what is needed.

2.2.4 Gerrit

Gerrit is a web-based code review system built on top of the Git platform by Google. It offers a platform to review and comment on code changes. All changes need to be approved by other people by scoring their change on a scale from -2 to +2 (Gerrit, n.d.-b, n.d.-c).

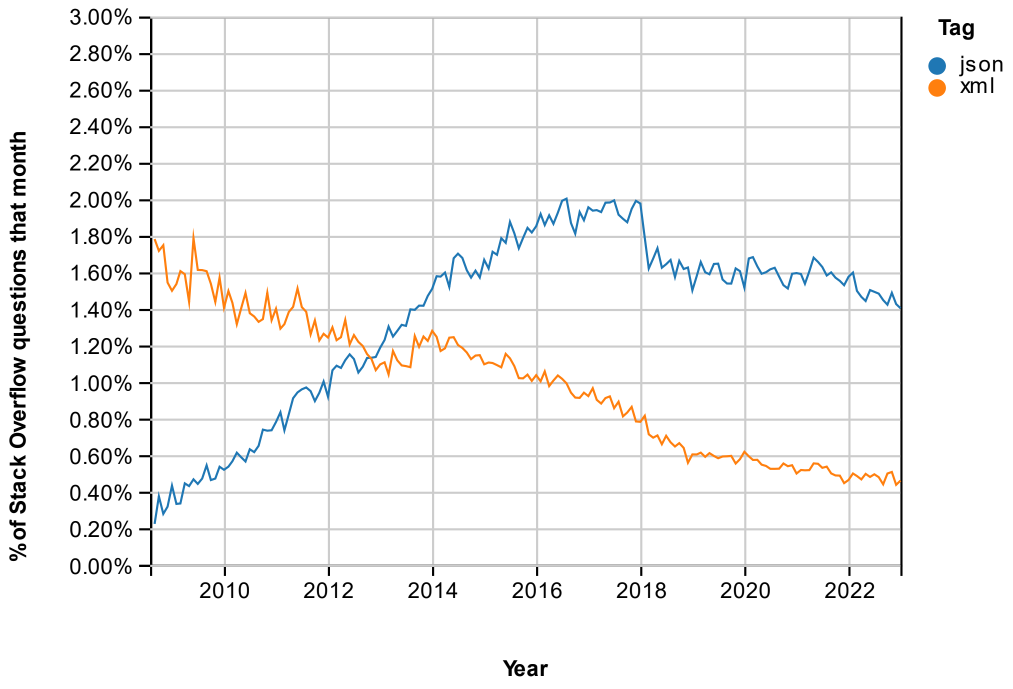
Gerrit can also be used as an all-in-one solution for Git and authentication, it can serve Git repositories and restrict access to those. By default, Gerrit allows read access to anonymous users, but that can be taken away if needed. Gerrit offers a high amount of customization for the server and allows the expansion of the features using plugins (Gerrit, n.d.-a).

Gerrit should be considered as an option for authenticating users, but should not be used exclusively for that. It is first and foremost a code review platform that makes the users stick to linear code commits through mandatory reviews.

2.3 JSON format

JavaScript Object Notation (JSON) is a data interchange format, which has been standardized by the Internet Engineering Task Force (IETF) as the Request For Comments (RFC) 8259, Ecma International, and the International Standards Organization (ISO) standard 21778:2017 (Bray, 2017; Ecma International, 2017; ISO/IEC 21778:2017, 2017). The earliest standard was published by the IETF as the RFC 4627 in July 2006 (Crockford, 2006), but according to the first specifier of JSON, Douglas Crockford, the earliest usage of the JSON format could be traced back to 1996 by an employee at Netscape (YUI Library, 2011). The JSON format is often used for communicating data across the internet, so it is generated and/or sent from a client to a server. Since it has been standardized by so many organizations and is widely used on the internet, it should be used to transport data to and from the application. Statistics from the Stack Overflow website show the popularity of questions mentioning JSON being around 1.4 % percent on the 1st of March 2023 versus Extensible Markup Language (XML) being mentioned approximately 0.5 % of the time.

Figure 5 Percentage of questions mentioning JSON vs. XML (Stack Overflow, 2023)



The JSON format uses human-readable key-value pairs to represent values. Which can be seen in Figure 6 below.

Figure 6 An example of a JSON object

```
{
  "array": [
    1,
    2
  ],
  "boolean": true,
  "number": 123,
  "object": {
    "a": "a",
    "b": "b"
  },
  "string": "Hello World!"
}
```

Even though JSON is intended to be a data interchange language, it is used as a format for configuration languages, for example with npm and yarn, both of which are package managers for JavaScript projects (npm Inc., n.d.; Yarn, n.d.). This usage of JSON is not the best out there: it does not support any comments in the file and according to Thayne

McCombs, it has a low signal-to-noise ratio (SNR), which means that there are unnecessary symbols that are required, leading to worse readability for humans. Another points that McCombs points out, are the lack of support for multi-line strings, for example, shown in Figure 6, as well as the specification for number format, which should be an “arbitrary precision finite floating point numbers in the decimal notation”, according to them. (McCombs, 2018)

As shown in Figure 7, the multi-line string uses three quotation marks to wrap the string value. This method of multi-line strings comes from Tom's Obvious Minimal Language (TOML), another file format made specifically for configuration files. The code written in Figure 6 would not work, since the multi-line string is not supported in JSON.

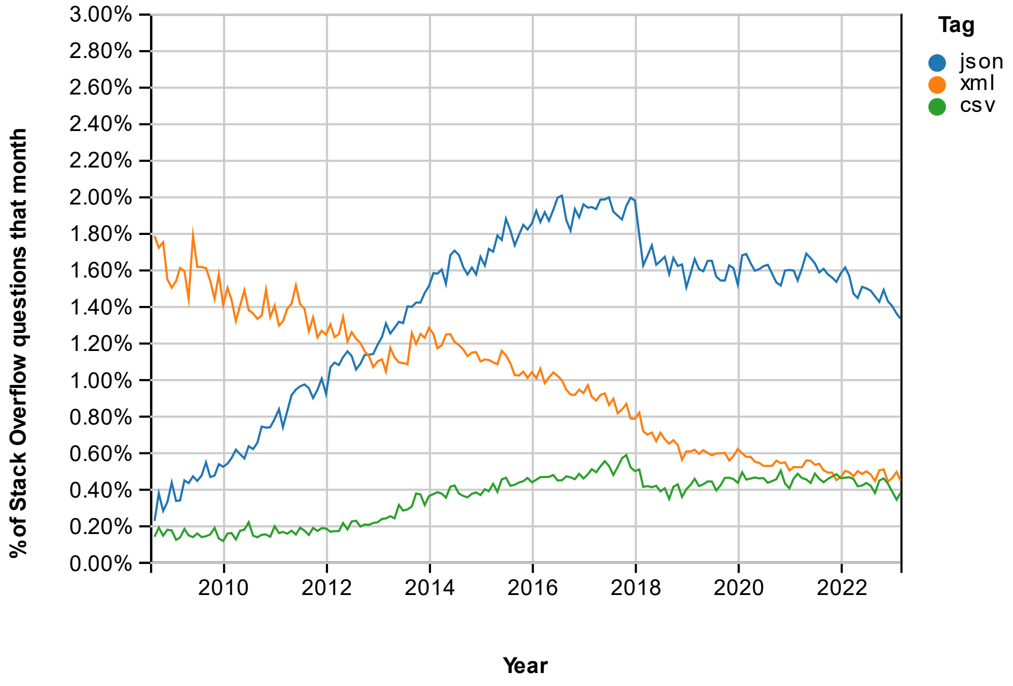
Figure 7 A proof-of-concept of a multi-line string in JSON

```
{
  "string": """
    Hello World!
    Hello World again
  """
}
```

2.4 JSON vs XML vs CSV

The tools used at the commissioner use the JSON format, but it should be explored how it compares to other data formats, such as the aforementioned XML and comma-separated values (CSV). When looking at the insights of Stack Overflow that include a comparison between JSON, XML, and CSV, it can be seen that CSV is the least popular tag in the comparison in Figure 8 below, falling just under 0.4 % of the questions.

Figure 8 Comparison between JSON, XML, and CSV



In the XML for Dummies book (Dykes & Tittel, 2011, p. 11), it is said that XML uses tags to represent values, as shown in Figure 9 below. It needs to open with a start-tag, has a value in the middle, and finally ends in an end-tag.

Figure 9 Representation of an XML element

```
<tag>value</tag>
```

XML offers the possibility to use a schema to have constraints when writing an XML file, which JSON does not offer by itself. The schema is not limited to a single type, but the World Wide Web Consortium recommends (World Wide Web Consortium, 2012) XML Schema Definition (XSD) to be used. According to Møller and Schwartzbach (Møller & Schwartzbach, 2006), there is a big general problem with the schema: its complicatedness in the specification. The specification is hundreds of pages long, so non-technical people would not read through it all just to write a simple schema.

According to Walker (Walker, 2020), a benefit of using XML is its support for other character encoding standards than UTF-8. This, however, should not be taken as a major benefit, since according to W3Techs (Q-success, 2023), overall 97.9 % of websites use UTF-8 as their character encoding standard and has been adopted and recommended by the IETF in RFC 2277 (RFC 2277:1998, 1998, p. 3) back in 1998.

JSON does not have a standardized schema, but there have been drafts towards that since 2009 (Internet Engineering Task Force, n.d.) with the latest official draft being from 2017 and it expired on the 17th of October 2017. An organization called JSON Schema (JSON Schema, n.d.-b) has developed the specification as an open-source project, with the latest specification being from 2020. One of the developers in the organization, Jason Desrosiers (Desrosiers, 2022), wrote in a blog post that the project has decoupled itself from the official IETF draft process, so it is up to the future is JSON Schema becomes an official standard in the future.

CSV files, however, are text files, which contain data records. CSV is not fully standardized, so the implementation varies per program and implementation. There is a proposal for a specification for CSV, the RFC 4180, and the RFC 7111, which attempted to create a standard but have not proceeded since 2014 (RFC 4180:2005, 2005; RFC 7111:2014, 2014). So using CSV files comes with the responsibility of finding compatibility between platforms and tools since there is no definite standard.

Since CSV files are essentially text files, data types cannot be defined in the file or a schema beforehand. This leaves the parser to interpret the types of data when ingesting or modifying the data. The header row containing the names of the fields is optional, so parsers need to consider that as well. Since CSV files contain data records, it would be more suitable to compare them to databases. In that comparison, CSV files do not offer any kind of encryption for the file itself (data can always be encrypted), so it should not be used there either.

Databases can utilize Transparent Data Encryption (TDE) to encrypt and decrypt the data in real time before the data is stored on a hard drive or read and sent forwards from the database. This technology does not encrypt or decrypt data in transport or when in use. TDE is done transparently, as the name suggests, so the users will not notice it. (Fauna, 2020) According to CSVLoader (CSVLoader, n.d.), “You cannot encrypt a CSV file directly without using a third-party program”. This means that without a third-party tool, all the data is unencrypted and in plaintext for anyone to read. This leaves the door open for hackers to read the data if they get access to the hard drive or the system. This is mitigated on databases if they use TDE since the data is then encrypted and unreadable without the encryption key.

To conclude the comparison between these three different file formats, JSON and XML could be comparable, but XML schemas would require more work and thought put into them. JSON does support schemas, but there is no official standard for those. The JSON Schema website (JSON Schema, n.d.-a) lists implementations that work the draft-06 or later, so when using the schema, it is up to the developers to find an implementation that would work for them. CSV files should be treated as text files, and not used for configuration data due to the lack of a standard implementation, or especially as a database due to the lack of security without any third-party tools.

2.5 Testing

The software will be tested with a testing framework called JUnit. This testing framework allows unit tests to be written and tested in real-time while writing the software itself to ensure that the software works as intended and does not produce any unwanted behavior.

Testing software, while it is developed, can reduce the time that is taken to write the software, since the tests and testing frameworks can spot errors consistently and perhaps find errors that the developers were not aware of, thus leading to reduced costs for businesses to develop the wanted software.

Since this software uses JSON files, they should be tested for integrity as well, but this will be achieved by a Java library made by Google, called Gson. Gson allows for the deserialization of JSON strings and files, which can be parsed into predefined Java classes, ensuring that the data exists in the file provided, and is compliant with the class and data type it is supposed to be. (Google, 2015/2023)

3 Project handling and reporting

The thesis will utilize an incremental and prototyping project model, since the project this thesis concerns does not have a lead developer looking after it but does have clear goals for what needs to be achieved. This means that the project will create an initial prototype of the software, and then gradually evolve it until it meets the requirements set by the commissioner and the software works as intended.

The thesis will progress in real-time when writing code snippets for proof of concepts (PoCs), hence there is no need to document what code has been written and why since that will be written in the code itself as comments to explain the reasoning and/or functions of it.

Since the commissioner handles confidential data, the code and information provided will be made for general use instead of basing it on real data or use cases.

4 Implementation of the project

The project requires Java 11 runtime to be installed to run the final executable.

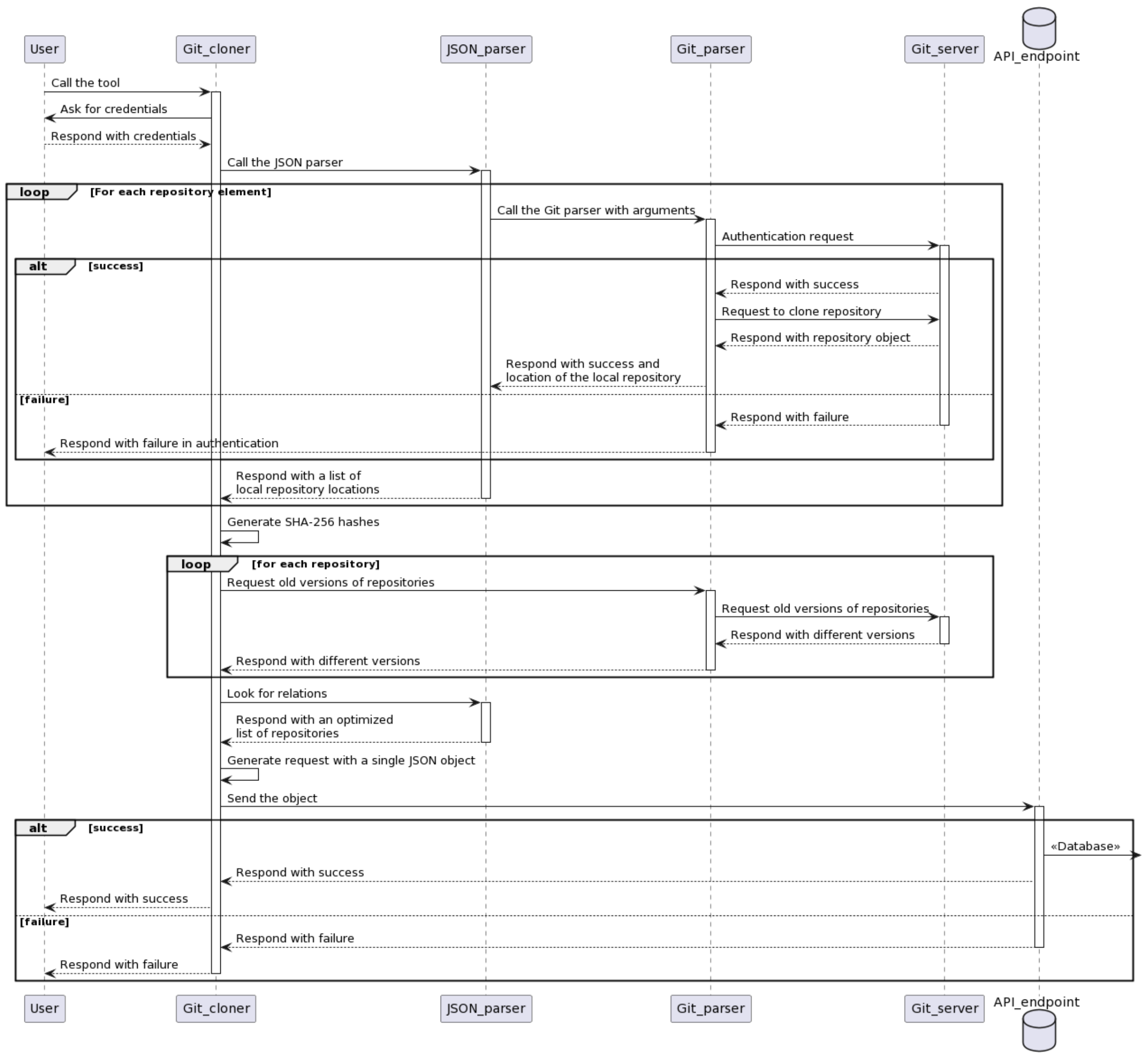
As an example, the project has a JSON file that lists the repositories that need to be pulled, shown in Appendix 2. The JSON object inside it has an array of repositories, with a key called “repositories”. That array contains objects inside it, defined by keys that have values of “ID_<number>”. Those objects are individual repositories, each having data unique to them, but all of them share the same data structure.

As this thesis focuses on Git interaction, the only data inside the objects that are important would be the “Name”, “Destination” and “URL” key-value pairs. The name of the repository could match the one in the URL path, and the destination of the repository should be the “Destination” value if that path is valid in the system.

The data that will be sent to the database will be normalized in the database schema, but this thesis will not go into detail about how that is done. The endpoint however would require normalized data to feed it to the database, so all different types of values need to be taken into account when creating the final payload, most importantly how the endpoint handles null values since those can be used as a valid value.

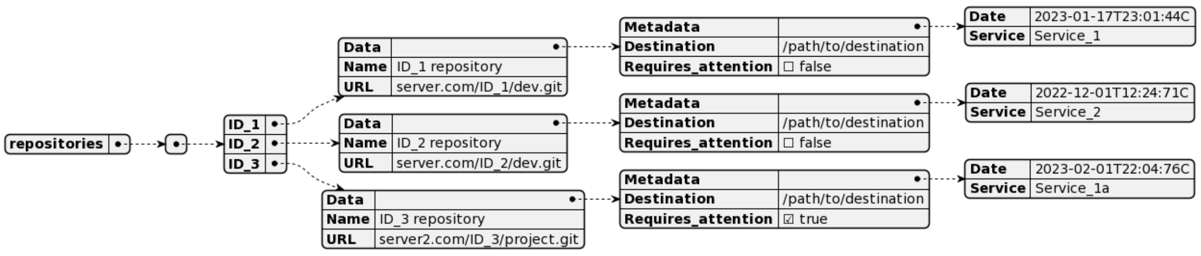
The general lifecycle of the tool and each component can be seen in Figure 10.

Figure 10 Sequence diagram of the tool


User
Git_cloner
JSON_parser
Git_parser
Git_server
API_endpoint
Call the tool
Ask for credentials
Respond with credentials
Call the JSON parser
loop [For each repository element]
Call the Git parser with arguments
Authentication request
alt [success]
Respond with success
Request to clone repository
Respond with repository object
Respond with success and location of the local repository
[failure]
Respond with failure
Respond with failure in authentication
Respond with a list of local repository locations
Generate SHA-256 hashes
loop [for each repository]
Request old versions of repositories
Request old versions of repositories
Respond with different versions
Respond with different versions
Look for relations
Respond with an optimized list of repositories
Generate request with a single JSON object
Send the object
alt [success]
«Database»
Respond with success
Respond with success
[failure]
Respond with failure
Respond with failure

The visualization of the example.json file can be seen in Figure 11.

Figure 11 Visualization of example.json


repositories
ID_1
ID_2
ID_3
Data
Name ID_1 repository
URL server.com/ID_1/dev.git
Data
Name ID_2 repository
URL server.com/ID_2/dev.git
Data
Name ID_3 repository
URL server2.com/ID_3/project.git
Metadata
Destination /path/to/destination
Requires_attention ☐ false
Metadata
Destination /path/to/destination
Requires_attention ☐ false
Metadata
Destination /path/to/destination
Requires_attention ☑ true
Date 2023-01-17T23:01:44C
Service Service_1
Date 2022-12-01T12:24:71C
Service Service_2
Date 2023-02-01T22:04:76C
Service Service_1a

4.1 Pulling multiple protected repositories

After the initial design has been created for the project, it should be explored how repositories, that are private, hidden, and require authentication to access, can be pulled to the local machine using Java.

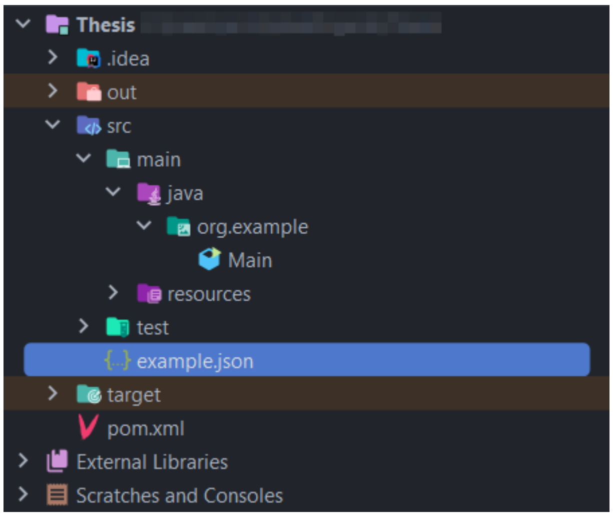
To parse JSON in Java, the project needs to be initialized as a Java Maven project, which gives it the capability to use the Maven project management tool. This can be achieved in JetBrains IntelliJ IDEA integrated development environment (IDE) by creating a new project, choosing Java as the language, and selecting Maven as the build system. This creates a pom.xml file in the project root directory, which is the project configuration file. Then the pom.xml file needs Gson library dependency written to the “dependencies” field as shown in Figure 12.

Figure 12 Maven dependency for Gson library in pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.10.1</version>
    </dependency>
</dependencies>
```

To read the JSON file, which for the sake of this thesis, is in the “src” folder of the project as highlighted in blue in Figure 13.

Figure 13 Location of example.json file in the project



This file can be read by the Gson library with the code depicted in Figure 14 below and will produce an output the same as the file shown in Appendix 2. The code in Figure 14 has comments to describe some of the functions it uses.

Figure 14 Java code to read the JSON file in Main.java

```java
package org.example;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

import java.io.Reader;
import java.nio.file.Files;
import java.nio.file.Paths;

public class Main {
    public static void main(String[] args) {
        try {
            // Create a reader for the example.json file
            Reader reader =
Files.newBufferedReader(Paths.get("src/example.json"));
            // Parse the example.json file to a JsonObject
            JsonObject jsonObject =
JsonParser.parseReader(reader).getAsJsonObject();
            // Create a Gson object for "pretty printing"
            Gson gson = new GsonBuilder().setPrettyPrinting().create();
            // Print the JsonObject using the gson object
            System.out.println(gson.toJson(jsonObject));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

To get each of the URLs on the file, the JsonObject needs to be parsed further with Gson. Continuing from the previous Figure 14, the JsonObject should be parsed with the `get` function from the Gson library to drill into the “repositories” element. That element will be assigned to a JsonObject called repo_elementAsJsonObject. That object’s entry set, the data inside the ID_1, ID_2, and ID_3 elements, will be looped through, and the URL field will be searched for, shown in Figure 15, and printed to the console with the expected output shown in Figure 16.

Figure 15 Mapping repository entries in Main.java

```
// Get the repository field from the JSON file and loop through it
for (JsonElement repo_element : jsonObject.get("repositories").getAsJsonArray()) {
    // Convert the repo_element into a JsonObject
    JsonObject repo_elementAsJsonObject = repo_element.getAsJsonObject();
    // Get each of the entries from the repo_elementAsJsonObject as a key-value
       map
    for (Map.Entry<String, JsonElement> entry :
repo_elementAsJsonObject.entrySet()) {
        /*
         * Find the URL values inside each entry
         * This is done by getting the value from the map object,
         * converting it to a JsonObject and finding the "URL" field
         * and finally converting it to a String
         */
        String URL =
String.valueOf(entry.getValue().getAsJsonObject().get("URL"));
        System.out.println(URL);
    }
}
```

Figure 16 Console output of Figure 12

```
"server.com/ID_1/project.git"
"server.com/ID_2/dev.git"
"server2.com/ID_3/project.git"
```

As seen in Figure 16 above, the output matches the values in the URL fields from Appendix 2.

4.2 Authentication

Now that the program has the Git repository addresses, the next step would be to pull them to their desired destination folders. Since Git supports multiple different protocols for transporting data, there are multiple ways to authenticate a user via these protocols.

The easiest method would be HTTP, since that would only require a username and a password to be provided to the server, instead of a certificate or a key file for SSH. The HTTP protocol can also support OAuth tokens, but that requires configuration from the authentication provider since the token needs to be supplied either as the username or the password.

If there is a need to use SSH instead of HTTP authentication, the internet has plenty of instructions on how to create an SSH key pair. The Git provider GitHub only supports OpenSSH SSH keys and on the other hand, GitLab, a competitor to GitHub, still supports Rivest–Shamir–Adleman (RSA) SSH keys which JGit, a popular library implementing Git support for Java, supports as well, so choosing the correct commercial platform matters in this case because not all of them support the same SSH key formats (GitHub, n.d.-a; GitLab, n.d.). Jgit removed its dependency on a library called Java Secure Channel, JSch, which previously was used for authentication, and moved to its independent implementation, which does not have any concrete instructions for implementing it in the code, so it should be considered if the time investment is necessary to use SSH authentication instead of HTTP authentication, which has stayed the same despite the changed in dependencies in JGit (Eclipse Foundation, Inc., n.d.). The code for authenticating the user and cloning the repositories can be found in Appendix 2.

4.3 Verifying the files and generating file hashes

To ensure that the combined file does not lose the integrity of the individual files, file hashes should be created to ensure they stay the same from the Git repositories, all the way to their destination. Java has a built-in library that contains a function to create the SHA-256 hashes, and that should be used for that purpose. Those hashes that are created from the files should be appended to the combined JSON object using unique element names that are derived from the individual files, or appended to the individual files before they are combined in Chapter 4.6.

For example, the SHA-256 checksum for the example.json from the Appendix 2 file is “48a752b019b6609d3f7965d99321e9ac47ca2803b1e8c61fc0d6586b3376911a”, which should stay the same as long as the file is not modified. This was talked about more in-depth in chapter 2.4.

5 Retaining versioning

It should be explored if there is a need to retain the versioning when moving from Git to a database. Since the pure implementation of Git does not support any specific artifacts, packages, or releases, that can only be done with commits. These commits can be assigned an incremental value when crawling through them and sending that value as the version number to the database. This incremental versioning does not support versioning for Git branches. Commits can be further filtered down with tags, that would indicate different versions.

5.1 JGit

JGit offers some functions to crawl through the commit history and search by tags, but getting the state of the repository by a commit tag requires more work than just calling a single function.

Figure 17 Pseudocode for resetting a repository to a desired commit

```
#Call the function with a repository object
Function get_repo_by_commit(Repository repo, String commit_tag) {
    String commit_SHA =
repo.getTagByName(commit_tag).getCommitHash();

    #Reset the repository to the state at the commit
    repo.reset().mode("hard").setRef(commit_SHA);
}
```

In the figure above, the pseudocode requires a repository object and a commit tag to be fed as arguments. The tag is used to look for a commit made by that tag in the repository and get that specific commit's SHA hash. That hash will be used by the reset function found in Git, and therefore in JGit, to reset the repository to the state that it was when that commit was applied. This repository can be then considered as a previous version.

If these different versions would be used, they could be combined into a single JSON object with the structure depicted in Figure 18. Splitting the versions into objects inside an array makes the iteration of different versions easy since they can be iterated by using a simple `for each` loop in the software.

Figure 18 Pseudo example of versioning inside JSON

```
{
  "repositories": [
    {
      "ID_1": [
        {
          "V_1": {
            "SHA256": hash here...,
            Version 1 data here...
          }
        },
        {
          "V_2": {
            "SHA256": hash here...,
            Version 2 data here...
          }
        }
      ]
    }
  ]
}
```

6 Finding relationality

To reduce the total request size of the final JSON object, it should be explored if finding relations between different objects is worth the investment and processing cost and speed. If the object can be simplified to just three fields in one layer, containing the name, the name as well the ID of the related object, and the SHA256 hash as a confirmation field. To calculate the reduction in fields, the percentage decrease equation can be used:

$$r = percentage\ decrease\ when\ using\ relation$$

$$F1 = number\ of\ fields\ in\ an\ object > 3$$

$$F2 = number\ of\ field\ after\ relation = 3$$

The equation for percentage decrease is:

$$\frac{F1 - F2}{F1}$$

To use the equation above, if an object that has 10 fields can be simplified down to 3 fields, the equation for the reduction becomes:

$r = \frac{10-3}{10} * 100 = \frac{7}{10} * 100 = 70$, which will equal a reduction of fields by 70 %.

But the amount of fields does not matter, but the length of the total object gets simplified down to 3 fields. If the ID_1 object is taken from Appendix 2 and shown in Figure 19 below, that object has a total of 205 characters excluding the whitespace characters, such as a tab or a space.

Figure 19 ID_1 object from Appendix 2

```json
{
  "ID_1": {
    "Data": {
      "Metadata": {
        "Date": "2023-01-17T23:01:44C",
        "Service": "Service_1"
      },
      "Destination": "/path/to/destination",
      "Requires_attention": false
    },
    "Name": "ID_1 repository",
    "URL": "server.com/ID_1/dev.git"
  }
}
```

If the 3-field limit is applied to the figure above, the object would be turned into the one in Figure 20, with a total character length of 133. This is already a decrease of approximately 35 %.

Figure 20 Example of a simplified JSON object

```
{
  "ID_1": {
    "Name": "ID_1 repository",
    "Reference_ID": "ID_0",
    "SHA256":
"9197378093ddc68fa86df23dd0ad1064306a041cbdaa658b0b340a94d4890a55"
  }
}
```

6.1 Combining all the files into a single JSON object

Now that the tool has cloned the repositories to their destination folders, the configuration file or files can be processed and sent to the API endpoint. It depends on the API endpoint or points, and how the data should be sent, but in this thesis, the assumed endpoint is the only one that accepts all the data. Therefore, the configuration files should be merged into a single JSON object to be sent to the endpoint.

The configuration files can be combined with the code shown in Figure 21.

Figure 21 Code to combine JSON files

```
Gson gson = new Gson();
JsonObject combinedJson = new JsonObject();

JsonObject json1 = gson.fromJson(jsonString1, JsonObject.class);
JsonObject json2 = gson.fromJson(jsonString2, JsonObject.class);
JsonObject json3 = gson.fromJson(jsonString3, JsonObject.class);

for (Map.Entry<String, JsonElement> entry : json1.entrySet()) {
    combinedJson.add(entry.getKey(), entry.getValue());
}

for (Map.Entry<String, JsonElement> entry : json2.entrySet()) {
    combinedJson.add(entry.getKey(), entry.getValue());
}

for (Map.Entry<String, JsonElement> entry : json3.entrySet()) {
    combinedJson.add(entry.getKey(), entry.getValue());
}

String combinedJsonString = gson.toJson(combinedJson);
```

7 Testing in practice

Since this code requires a JSON file to be read from, the code should inform the user if that file is not provided, which will be done by the Java runtime in the form of a runtime exception if the code is surrounded by a try-catch clause.

7.1 Parsing JSON

Since this tool is intended for ingesting configuration data, the JSON parser should be tested with a file that has the same internal structure as the final, actual files will have. This ensures that the parser will find the right elements in the files and handle the data inside them as needed. As the data from the JSON file is parsed into Java data types, JUnit can be used for testing the output of each element to an expected value, as shown in Figure 22.

Figure 22 Test to assert the URL

```
String URL =
String.valueOf(entry.getValue().getAsJsonObject().get("URL")).replace("\"", "");
String expected_output = "server2.com/ID_3/project.git";

assertEquals(URL, expected_output);
```

That test assumes that the URL element in the test JSON file matched the value in \`expected_outout\`. This testing methodology can be applied to every element in the JSON file if the application grows more complex and has more functions for each element, testing the variable assignment as shown above, ensure that the file structure is correct, and each variable will be assigned the correct value.

7.2 Authentication

Testing the authentication is crucial for a server since it is crucial to not allow people accessing the server who do not need access to it. This includes people who just do not need access to the server since it does not concern their side of the work, hackers who try to steal data from your server, as well as people trying to access the server by accident, or web bots that scrape the internet looking for servers and web addresses.

This form of authentication testing should be left to the server, instead of the client, but it is possible to confirm that a client cannot access the server without approved credentials. This is done easily on the client side, just by trying to use incorrect credentials to authenticate the request to the server, and the expected response should be a failure to authenticate.

7.3 Versioning

A testing environment running Git should be created alongside a repository, where commits will be pushed with tags to test the functionality of retaining the versioning. At this point of testing, it should be tested how different kinds of commits affect the tool, for example, whether merging changes the versioning or not.

8 Summary

The first question regarding why Git is not suitable for hierarchical data can be concluded by the fact that Git is not a database system, and therefore should not be used as one. It does not retain any relationships between files or data, and a database should be used if that is needed. Then, is it possible to retain versioning from Git when moving to another solution? Yes, it is possible, but the pure implementation of Git does not have any artifacts like GitHub or GitLab, so commits and their tags are the best way to achieve this. Lastly, it should be explored on a project-by-project basis whether finding relations in data is worth the processing investment that depends on the complexity of the data set in question. For smaller projects that do not care about the investment for any reason, this might be a good investment.

The author gained knowledge starting from remembering the Java syntax to finding out how Git packages operate under the hood. The most interesting part of the research was the Git mailing lists regarding the security of SHA-1 hashes and the cyclic development of the SHA-256 implementation, and the reasons why it is not a high priority, at least until SHA-1 will be broken completely. JSON files are a common way for writing configuration files, but when looking at its technical capabilities, it is not among the best options available, due to it missing many common features, like comments.

This thesis offers reasons to switch from Git-based database solutions to proper databases and introduces some code that explains how that could be achieved. It also talks about how JSON files can be optimized to have some relationality, but that assumes that the destination of the JSON object is configured to support the same format of relationality. So far, the commissioner has been pleased with this thesis the results and findings of the thesis will be put into work at Fail-Safe IT Solutions Oy in some manner, and this thesis was the background research for that purpose.

References

Bray, T. (2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. https://www.rfc-editor.org/info/rfc8259

Chacon, S., & Straub, B. (2014). *Pro git: Everything you need to know about Git* (Second edition). Apress. https://git-scm.com/book/en/v2

Comer, D. (1979). Ubiquitous B-Tree. *ACM Comput. Surv.*, *11*(2), 121–137. https://doi.org/10.1145/356770.356776

Crockford, D. (2006). *The application/json Media Type for JavaScript Object Notation (JSON)* (Request for Comments RFC 4627). Internet Engineering Task Force. https://doi.org/10.17487/RFC4627

CSVLoader. (n.d.). *Is it possible to encrypt CSV file?* Retrieved 7 April 2023, from https://csvloader.com/csv-guide/how-to-encrypt-a-csv-file

Desrosiers, J. (2022, October 21). *Towards a stable JSON Schema | JSON Schema Blog*. https://json-schema.org/blog/posts/undefined/posts/future-of-json-schema

Dykes, L., & Tittel, E. (2011). *XML For Dummies*. Wiley. https://books.google.fi/books?id=7msUAZm1HcUC

Eclipse Foundation, Inc. (n.d.). *JGit/New and Noteworthy/5.8—Eclipsepedia*. Retrieved 17 February 2023, from https://wiki.eclipse.org/JGit/New_and_Noteworthy/5.8

Ecma International. (2017). *ECMA-404*. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

endoflife.date. (n.d.). *Java/OpenJDK End of Life*. Endoflife.Date. Retrieved 3 March 2023, from https://endoflife.date/java

Fauna. (2020, July 24). *How Does Database Encryption Work?* Fauna. https://fauna.com/blog/database-encryption

Gerrit. (n.d.-a). *Gerrit Plugins*. Retrieved 4 April 2023, from https://gerrit-documentation.storage.googleapis.com/Documentation/3.6.0/config-plugins.html

Gerrit. (n.d.-b). *How Gerrit Works*. How Gerrit Works. Retrieved 3 April 2023, from https://gerrit-review.googlesource.com/Documentation/intro-how-gerrit-works.html

Gerrit. (n.d.-c). *Working with Gerrit: An example*. Retrieved 4 April 2023, from https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrough.html

GitHub. (n.d.-a). *Generating a new SSH key and adding it to the ssh-agent*. GitHub Docs. Retrieved 17 February 2023, from https://ghdocs-prod.azurewebsites.net/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

GitHub. (n.d.-b). *Tags · git/git*. GitHub. Retrieved 23 January 2023, from https://github.com/git/git

GitLab. (n.d.). *Use SSH keys to communicate with GitLab | GitLab*. Retrieved 17 February 2023, from https://docs.gitlab.com/ee/user/ssh.html

Google. (2023). *Gson* [Java]. Google. https://github.com/google/gson (Original work published 2015)

Internet Engineering Task Force. (n.d.). *Diff: Draft-zyp-json-schema-01.txt—Draft-zyp-json-schema-04.txt*. Retrieved 5 April 2023, from https://author-tools.ietf.org/iddiff?url1=draft-zyp-json-schema-01&url2=draft-zyp-json-schema-04&difftype=--html

ISO/IEC 21778:2017. (2017). *ISO/IEC 21778:2017* (p. 6). https://www.iso.org/standard/71616.html

JSON Schema. (n.d.-a). *Implementations*. JSON Schema. Retrieved 5 April 2023, from https://json-schema.org/implementations.html

JSON Schema. (n.d.-b). *Specification*. JSON Schema. Retrieved 5 April 2023, from https://json-schema.org/specification.html

Laster, B. (2016). *Professional Git*. John Wiley & Sons, Incorporated. http://ebookcentral.proquest.com/lib/hamk-ebooks/detail.action?docID=4751486

Mäsak, C. (2021, October 18). *How is git commit sha1 formed*. Gist. https://gist.github.com/masak/2415865

McCombs, T. (2018, July 16). Why JSON isn’t a Good Configuration Language. *Lucidchart*. https://www.lucidchart.com/techblog/2018/07/16/why-json-isnt-a-good-configuration-language/

Møller, A., & Schwartzbach, M. (2006, February). *Problems with XML Schema*. https://cs.au.dk/~amoeller/XML/schemas/xmlschema-problems.html

Nemeth, E., Snyder, G., Hein, T. R., Whaley, B., & Mackin, D. (2017). *UNIX and Linux System Administration Handbook* (5th ed.). Pearson Education.

New Relic. (2021, August 5). *2022 State of the Java Ecosystem Report | New Relic*. https://newrelic.com/resources/report/2022-state-of-java-ecosystem

npm Inc. (n.d.). *Creating a package.json file | npm Docs*. Retrieved 30 January 2023, from https://docs.npmjs.com/creating-a-package-json-file

Oracle. (n.d.). *Oracle Java SE Support Roadmap*. Retrieved 15 February 2023, from https://www.oracle.com/java/technologies/java-se-support-roadmap.html

Q-success. (2023, April 3). *Usage Survey of Character Encodings broken down by Ranking*. https://w3techs.com/technologies/cross/character_encoding/ranking

RFC 2277:1998. (1998). *IETF Policy on Character Sets and Languages* (Issue 2277). RFC Editor. https://www.rfc-editor.org/info/rfc2277

RFC 4180:2005. (2005). *Common Format and MIME Type for Comma-Separated Values (CSV) Files* (Issue 4180). RFC Editor. https://www.rfc-editor.org/info/rfc4180

RFC 7111:2014. (2014). *URI Fragment Identifiers for the text/csv Media Type* (Request for Comments RFC 7111). Internet Engineering Task Force. https://doi.org/10.17487/RFC7111

Shah, S. (2022, June 23). *Merkle Trees and their application in Git*. A Blog about Our Findings and Musings. https://ieee.nitk.ac.in/blog/merkle-trees-and-their-application-in-git/

Spinellis, D. (2012). Git. *IEEE Software*, *29*(3), 100–101. https://doi.org/10.1109/MS.2012.61

Stack Overflow. (2023, March 1). *Stack Overflow Trends: JSON vs. XML*. https://insights.stackoverflow.com/trends?tags=json%2Cxml

Stolee, D. (2022, August 29). Git's database internals I: Packed object store. *The GitHub Blog*. https://github.blog/2022-08-29-gits-database-internals-i-packed-object-store/

Torvalds, L. (2007, February 27). *Re: Trivia: When did git self-host?* https://marc.info/?l=git&m=117254154130732

Vermeer, B. (2020, February 5). *64% of developers report that Java 8 remains the most often used release | Snyk*. https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release/

Walker, A. (2020, February 25). *JSON vs XML – Difference Between Them*. https://www.guru99.com/json-vs-xml-difference.html

World Wide Web Consortium. (2012, April 5). *W3C News Archive: 2012 W3C*. https://www.w3.org/News/2012#entry-9412

Yarn. (n.d.). *Yarn*. Yarn. Retrieved 30 January 2023, from https://classic.yarnpkg.com/en/docs/package-json/

YUI Library (Director). (2011, August 29). *Douglas Crockford: The JSON Saga*. https://www.youtube.com/watch?v=-C-JoyNuQJs

Appendix 1: Thesis Data Management Plan

This thesis will not gather any research or personal data in any matter.

This thesis is commissioned by Fail-Safe IT Solutions Oy and the ownership of the thesis remains with the author of this thesis.

Appendix 2: Example JSON file of repositories

```json
{
  "repositories": [
    {
      "ID_1": {
        "Data": {
          "Metadata": {
            "Date": "2023-01-17T23:01:44C",
            "Service": "Service_1"
          },
          "Destination": "/path/to/destination",
          "Requires_attention": false
        },
        "Name": "ID_1 repository",
        "URL": "server.com/ID_1/dev.git"
      },
      "ID_2": {
        "Data": {
          "Metadata": {
            "Date": "2022-12-01T12:24:71C",
            "Service": "Service_2"
          },
          "Destination": "/path/to/destination",
          "Requires_attention": false
        },
        "Name": "ID_2 repository",
        "URL": "server.com/ID_2/dev.git"
      },
      "ID_3": {
        "Data": {
          "Metadata": {
            "Date": "2023-02-01T22:04:76C",
            "Service": "Service_1a"
          },
          "Destination": "/path/to/destination",
          "Requires_attention": true
        },
        "Name": "ID_3 repository",
        "URL": "server2.com/ID_3/project.git"
      }
    }
  ]
}
```

Appendix 3: Code for Git parser and authentication

```
public class Git_parser {
    protected static void parse(String URL, Path path) {
        try {
            String username;
            char[] password;
            CloneCommand cloneCommand = Git.cloneRepository();

            // Get credential from console
            username = getUsername();
            password = getPassword();

            System.out.println(username + Arrays.toString(password));
            // Set the URL
            cloneCommand.setURI(URL);
            // Create a CredentialProvider with username and password
            cloneCommand.setCredentialsProvider(new
UsernamePasswordCredentialsProvider(username, Arrays.toString(password)));
            // Set the destination path
            cloneCommand.setDirectory(path.toFile());
            // Call the function
            cloneCommand.call();
        } catch (GitAPIException e) {
            throw new RuntimeException(e);
        }
    }

    private static String getUsername() {
        String username;
        System.out.println("Enter your Git username: ");

        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        try {
            username = reader.readLine();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return username;
    }

    private static char[] getPassword() {
        char[] password;
        // DOES NOT WORK in IDE consoles
        //
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Console.htm
l
        Console console = System.console();

            try {
                password = console.readPassword("Enter your Git password: ");
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        return password;
    }
}
```