Design and Implementation of a Version Control System for Open Data Modelling Projects

MASTER THESIS

Martin Buchalik

Submitted on 15 August 2022

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

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

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 15 August 2022

License

Erlangen, 15 August 2022

Abstract

Many modern software applications and research projects depend on the ability to access high-qualitative data sources. Even though there is already a large number of openly available data sets, such data sets are often hard to (re)use due to various barriers such as incomplete documentation, wrong or missing values, and more. To address these barriers, the JValue Project has been established by the Professorship of Open Source Software at Friedrich-Alexander-Universität Erlangen-Nürnberg. The goal of the JValue Project is to "make open data easy, safe, and reliable".

In the context of the JValue Project, numerous software applications are developed which, among others, allow to explicitly define the structure and further meta information of openly available data sets. However, it is currently neither possible to collaborate with other individuals on such data source configurations, nor is it possible to retrace the historic development that led to the current state of a particular configuration.

To build a basis to address these issues, a Version Control System shall be developed, which makes it possible to store, retrieve, and compare revisions of files containing data source configurations and related information. This thesis presents a concept of such a system, and evaluates this concept by implementing a prototype showing its feasibility.

As a result of this thesis, it is now possible for other applications developed in the context of the JValue Project to access, create, and compare revisions in order to provide advanced collaboration and versioning features to end users.

Contents

1 Introduction **1**

2 Fundamentals **5**
2.1 Context: The JValue Project 5
2.1.1 Data Engineering Workbench 6
2.1.2 JValue Hub 6
2.2 Version Control Systems 7
2.2.1 Basic terminology 7
2.2.2 Diff 9
2.2.3 Core features of a VCS 9
2.2.4 Types of Version Control Systems 11
2.3 Trees and graphs 13
2.3.1 Trees 13
2.3.2 Graphs 16

3 Requirements **19**
3.1 Source of requirements 19
3.1.1 Adaptation of the Core Operations 19
3.2 Requirements format 20
3.3 Metric to summarize requirements completion 21
3.4 Functional requirements 21
3.5 Non-functional requirements 25

4 Conceptual design **27**
4.1 Comparison of CVCSs and DVCSs 27
4.2 Reusing existing general-purpose systems 28
4.2.1 Reusing user interfaces 28
4.2.2 Reusing HTTP APIs 28
4.3 Logical Model of the Hub VCS 29
4.3.1 Initial Logical Model 29
4.3.2 Model of files, folders, and delta 32
4.3.3 Model of the commit history 35

4.3.4 Updated Logical Model 40

5 Architecture **43**
5.1 Foundation of final design 43
5.2 Backend 43
5.3 Clients 44
5.3.1 User-facing clients (UIs) 45
5.3.2 System test suite and Importer 46
5.4 Final system structure 49

6 Implementation **51**
6.1 Database schema 51
6.1.1 Storage of files and folders 51
6.1.2 Storage of commits, repositories, and more 55
6.2 Creation of commits 59
6.3 Retrieval of diffs 64
6.3.1 Basic request processing 64
6.3.2 Detection of moved files 67
6.3.3 Result format 72

7 Evaluation **75**
7.1 Used notation for endpoints 75
7.2 Evaluation of functional requirements 75
7.3 Evaluation of non-functional requirements 91
7.4 Summary 92

8 Conclusion **95**

Appendix **97**
Mockups of DEWB and JValue Hub 99

References **103**

Acronyms

API Application Programming Interface

CLI Command Line Interface

CVCS Centralized Version Control System

DAG Directed Acyclic Graph

DEWB Data Engineering Workbench

DVCS Distributed Version Control System

ETL Extract, Transform, Load

HTTP Hypertext Transfer Protocol

ID Identifier

IDE Integrated Development Environment

JSON JavaScript Object Notation

ODS Open Data Service

UI User Interface

URL Uniform Resource Locator

UX User Experience

VCS Version Control System

1 Introduction

> *"There is a long history of governments, businesses, science and citizens producing and utilising data in order to monitor, regulate, profit from, and make sense of the world. Data have traditionally been time-consuming and costly to generate [...]" (Kitchin, 2014)*

Members of modern societies - companies, scientists, and more - often have a great need for high-qualitative data. This is also represented in the term "Information Society", which describes societies that highly depend on knowledge gained from information (Wessels, Finn, Wadhwa & Sveinsdottir, 2017).

Being able to access large amounts of data can have a major positive impact on efficiency and innovation. Thus, in recent years, especially governments have decided to make data available to the public. (Huijboom & van den Broek, 2011)

Such openly available data is commonly referred to as "Open Data". More precisely, the term "Open Data" describes the praxis of making data "accessible, understandable and open to reuse" (Wessels et al., 2017).

According to this definition of Open Data, it is not sufficient to publish raw[1] and undocumented data. However, according to a survey asking for the biggest barriers regarding Open Data in the context of governmental Open Data strategies, multiple factors were found that explicitly mention issues with the aforementioned aspects. Two of the "top 10 barriers" are "Limited user-friendliness/info overload" and "Lack of standardisation of open data policy". Also, it has been mentioned that government data often has a low quality. (Huijboom & van den Broek, 2011)

In order to erase those and additional barriers in the context of Open Data, the "JValue Project" has been started by the Professorship of Open Source Software at Friedrich-Alexander-Universität Erlangen-Nürnberg. The goal of the JValue Project is to "make open data easy, safe, and reliable".

One major software application developed in the context of the JValue Project

[1] We define "raw data" as data that has not or barely been processed and thus may include missing values, errors, and the like.

is the "Open Data Service" (ODS). The ODS provides features to fetch data from multiple sources, perform "cleansing" and quality assurance operations, and finally to allow access to this enhanced data using a single and well-documented API. At the time of writing this thesis, configurations of the ODS were performed individually, meaning that every user of the ODS would create their own settings and typically keep them for themselves, instead of sharing the settings and being able to reuse already existing ones.

This strategy might work well for smaller projects and few data sources. However, for larger projects with many data sources, being able to reuse and extend already existing settings definitions, as well as collaborating with other data scientists, software developers, and the like, will most likely greatly increase the productivity of users of the ODS. Such a crowd-sourcing approach may lead to the creation of communities, which could experience benefits similar to the ones found in Open Source Software communities.

Because of this, plans are made to develop a collaboration platform, the "JValue Hub". In its essence, the JValue Hub will follow the ideas of platforms like GitHub[2] or GitLab[3], which are platforms that are commonly used by software developers to collaborate on software code. The major difference between these platforms and the JValue Hub is that the JValue Hub will explicitly be tailored to the needs of data scientists and developers who want to collaborate on and share models of Open Data sources. These models can eventually be used by the ODS to retrieve, "clean up", and enhance the data present in the respective Open Data source.

The goal of this thesis is to develop a core functionality of the JValue Hub: Version Control. Version Control is commonly used in the field of software development for "tracking and managing [of] revisions" (Loeliger & McCullough, 2012). A Version Control System (VCS) typically allows users to access historical revisions of source code and explore which changes have been made at which point in time (Loeliger & McCullough, 2012).

This thesis contributes to the JValue Project by

1. presenting a concept to support the JValue Hub with a web-based VCS
2. evaluating the concept by implementing a prototype showing its feasibility

[2]https://github.com, accessed on June 21, 2022
[3]https://gitlab.com, accessed on June 21, 2022

This thesis is structured as follows:

- Chapter 2 describes the most important concepts and terms used in this thesis, and lays out the context in which the novel system will be developed.
- Chapter 3 defines functional and non-functional requirements that shall be met by the final system.
- Chapter 4 explains important theoretical concepts needed for the development of the new VCS.
- Chapter 5 provides an overview of the technical structure of the novel system.
- Chapter 6 highlights interesting and particularly important parts of the final implementation.
- Chapter 7 evaluates the final implementation of the novel VCS against the functional and non-functional requirements defined in chapter 3.
- Chapter 8 provides a short summary of the previous chapters.

2 Fundamentals

This chapter explains the most important concepts and terms used in this thesis. Furthermore, this chapter lays out the context in which the novel system will be developed.

2.1 Context: The JValue Project

As mentioned in chapter 1, the goal of the JValue Project is to "make open data easy, safe, and reliable". To achieve this goal, multiple complex software applications, as well as related research projects, have been established and are actively being worked on.

The core components of the JValue Project are[1]:

- The "Data Engineering Workbench", and the "JValue Hub", which will be described in more detail in sections 2.1.1 and 2.1.2.
- The ODS, which provides features to load, "cleanse", and access data from multiple sources.
- A framework (meta-model) for modelling data sources and eventually of ETL pipelines.
- A compiler to make it possible to transform an ETL model to configurations of various runtimes.
- A "Cloud Service" to provide configured ETL runtime instances to clients.

The VCS developed in this thesis will be connected to multiple of the aforementioned components of the JValue Project. In the following, two software application projects are presented that are connected the closest: The "Data Engineering Workbench", and the "JValue Hub".

[1]The list of "core components" only reflects a fraction (the most important "pillars") of the software applications built in the context of the JValue Project.

2.1.1 Data Engineering Workbench

At the time of writing this thesis, a new software application called "Data Engineering Workbench" (DEWB) was under development. Conceptually, the DEWB is similar to an Integrated Development Environment (IDE) such as the Eclipse IDE[2] or Xcode[3]. The main goal of the DEWB is to support users who want to create data source configurations for the ODS by providing user-friendly editing capabilities, error prevention, previewing functionalities, and more.

The DEWB is a completely web-based editor, meaning that it can be used in any modern web browser, but it is not meant to be installed as a standalone software application on a user's computer. This approach makes it especially easy for new users to quickly try out the DEWB, since no complex set-up is required to run the application. But it also brings benefits to experienced "power users", such as being able to work on (and switch between) any computer that is connected to the internet, always using the most recent version of the application and thus never having to manually deal with software updates, and having a consistent editing experience across all supported devices.

One of the most important features of the DEWB is the ability to preview which "effect" a data source configuration has: When the user is in the process of creating a data source configuration, the DEWB will display a preview of the data fetched using the configuration that is currently being edited. This quick feedback allows the user to determine possible problems and also to get a better understanding of the influence certain settings have. For this purpose, being able to load data from a remote server is necessary, which is why the DEWB is not meant to be used offline.

2.1.2 JValue Hub

As described in chapter 1, plans are being made to develop a platform called "JValue Hub". This platform is supposed to be a central place for sharing configurations of the ODS. Users are encouraged to collaborate in order to provide configurations for new data sources, to enrich already existing ones, to discuss problems, and more. This is similar to the way developers collaborate in Open Source Software communities.

The JValue Hub will be connected to the DEWB: If a user wants to work on a configuration, he or she can simply launch the DEWB. Once the work is finished, it will be possible to feed the changes back to the JValue Hub, i.e. to update the current version found in the JValue Hub through the DEWB.

[2]https://www.eclipse.org/ide/, accessed on June 21, 2022
[3]https://developer.apple.com/xcode/, accessed on June 21, 2022

In order to guarantee that it is always possible to understand which changes were made, all historic versions must be archived and it must be possible to determine the changes made between two of such "snapshots". Otherwise, it might become hard for users to follow the reasoning behind certain changes, which in turn might lead to less trust in the quality and integrity of such configurations. This is why a VCS is of high importance for the success of the JValue Hub.

Since the VCS built in this thesis will later become a central part of the JValue Hub, we call this VCS the *JValue Hub VCS* or just *Hub VCS*.

2.2 Version Control Systems

As described in chapter 1, VCSs are typically used in software development to allow users to precisely track the changes made to one or multiple files. One important feature of a VCS is the ability to access historic revisions of the source code under development (Loeliger & McCullough, 2012). VCSs can simplify and speed up software development processes, especially when collaborating with other developers and concurrently working on the same files in a software project (Otte, 2009).

2.2.1 Basic terminology

Otte (2009) and Sink (2011) explain important terms that will be used throughout this thesis. The following explanations are based on their definitions.

Typically, software projects are organized into one or more *repositories*. A repository contains all files and folders, as well as all historic revisions. Sink (2011) describes repositories using an "equation":

$$repository = filesystem * time$$

When a user wants to edit files, he or she first needs to perform a *checkout* of a specific revision. A checkout creates a *working copy* of the files and folders of a revision specified by the user. The user can then start performing changes to the working copy, independently of other users. It is important to note that a checkout is performed for exactly one specific revision. One could say that the user checks out a "snapshot" from the history of the repository.

Often, users just want to check out the latest revision from a repository in order to base their work on the most recent version. This would require these users to frequently look up the identifier of the latest revision. To make this process easier,

some VCSs provide additional commands to check out the latest revision without having to specify an exact revision identifier. Often, the latest revision is referred to as the *head*. Additionally, users can create named references to individual revisions. Such a named reference is called *tag*. A tag can, for instance, be used to mark an important version.

When the user has been working on changes in the working copy and has finalized them, he or she needs to create a new revision in order to persist these changes in the repository. Such a single new revision is called a *commit*. A commit does not only contain information about the modifications of the working copy, but may also contain metadata: Typically, users can provide a *commit message* to explain what a particular commit has changed compared to the previous version. Also, many VCSs store the creation date and time of a commit.

When one user has created a new commit, other users need to perform an *update* if they want to apply these changes to their own working copy. An update is equivalent to a checkout if no changes have been applied to the working copy. However, if the working copy has been modified prior to the update, it becomes necessary to *merge*[4] the changes of the incoming commit to the ones of the working copy. If an automatic merge fails because the incoming commit contains changes that are not compatible with the ones from the working copy, a *conflict* occurs. For instance, in many VCSs a conflict will occur when the exact same line in a particular file has been modified both in the working copy and in the incoming commit. A VCS may then prompt the user to resolve the conflict manually.

When a developer works on a new feature, he or she might want to commit smaller, potentially unfinished or broken changes to the repository. This can especially happen when testing a potential solution for a problem, or when trying out a new way to fix a certain issue. In such cases, every commit can hinder other developers from continuing their work if they perform an update of their working copy with a potentially faulty revision. Because of this, when starting work on a new set of changes, developers often create a new *branch*. Commits are always pushed to one specified branch. Later, commits from one branch can be *merged* into another branch. One way to work with branches is to define a "Main Branch" where the latest stable revision can be found, and many "Feature Branches"[5]. In this particular way of organizing a repository, once a certain feature is finished on a Feature Branch, it will get merged into the main branch.

In some cases, developers want to copy an entire repository in order to work completely independently of the maintainers of the original repository. Such an

[4]It should be noted that the term "merge" is used in multiple ways in this thesis. Apart from merging an incoming commit to the working copy when performing an update, the term is also used when combining the changes from one branch into another.

[5]See https://martinfowler.com/bliki/FeatureBranch.html, accessed on June 21, 2022

action is often referred to as a *fork*[6]. It is important to note the differences between a branch and a fork: Branches are part of a repository. A repository may contain an arbitrary number of branches. When creating a new branch, it gets added to the repository and points to a particular revision. On the other hand, forks are copies of repositories. Thus, when creating a fork, the original repository is not modified.

2.2.2 Diff

The term *diff* is defined by Otte (2009) as "the changes between two revisions of a file". This implies that a diff can only be calculated for single files, and that a diff must always compare historic versions of the **same** file. In this thesis, we alter this definition: *A diff represents the changes (differences) between two files, or even between two directories.* When calculating the diff between two directories, we expect the diff to be performed recursively so that sub-directories are also compared, as well as their sub-directories and so on. Thus, it is possible to say "the diff between file A and file B" or "the diff between folder A and folder B". It is not necessary that "file A" and "file B" are historic revisions of one particular file. Instead, they may be two arbitrary files the user wants to compare.

Additionally, we say that a diff between two commits represents the differences between the files and folders pointed to by these two commits.

2.2.3 Core features of a VCS

The exact list of supported features differs between various VCSs. However, there are certain basic operations that should exist in every VCS in order to provide essential features needed for version control. Sink (2011) defines a set of 18 basic operations that "could be considered the core concepts of version control". These features are listed in table 2.1. In this thesis, we call the operations listed in this table "Core Operations". For instance, *Core Operation 1 (Create)* refers to the first operation listed in table 2.1.

[6]The term "fork" is used in multiple version control platforms, see for example https://docs.github.com/en/get-started/quickstart/fork-a-repo or https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html (both links accessed on June 21, 2022).

Number	Title	Description
1	Create	"Create a new, empty repository."
2	Checkout	*See section 2.2.1*
3	Commit	*See section 2.2.1*
4	Update	*See section 2.2.1*
5	Add	"Use the add operation when you have a file or directory in your working copy that is not yet under version control and you want to add it to the repository."
6	Edit	"Modify a file."
7	Delete	"Delete a file or directory."
8	Rename	"Rename a file or directory."
9	Move	"Move a file or directory. Use the move operation when you want to move a file or directory from one place in the tree to another."
10	Status	"List the modifications that have been made to the working copy."
11	Diff	"Status provides a list of changes but no details about them. To see exactly what changes have been made to the files, you need to use the diff operation."
12	Revert	"Undo modifications that have been made to the working copy."
13	Log	"Show the history of changes to the repository."
14	Tag	*See section 2.2.1*
15	Branch	*See section 2.2.1*
16	Merge	*See section 2.2.1*
17	Resolve	"Handle conflicts resulting from a merge."
18	Lock	"Prevent other people from modifying a file."

Table 2.1: The 18 core operations of a VCS as defined by Sink (2011). The descriptions quoted here are also excerpts from Sink (2011).

2.2.4 Types of Version Control Systems

Otte (2009) distinguishes two types of VCSs: Centralized Version Control Systems (CVCSs) and Distributed Version Control Systems (DVCSs)[7]. This differentiation is made because of fundamental differences in the way repositories are stored, accessed, and collaborated on.

Centralized Version Control Systems

In a CVCS such as CVS[8] or SVN[9], a repository is always stored in only one single place. Checkouts are always performed against this central repository. (de Alwis & Sillito, 2009) Thus, one can see the central repository as a server (Otte, 2009).

Figure 2.1 shows how collaboration using a CVCS is organized: The history is only stored on the central server. Clients ("Computer A" and "Computer B") always only communicate with this central repository. Thus, every commit is always sent to the central server. It is not possible for clients to exchange commits between themselves. This also makes the central server a Single Point of Truth since it is the only place where the entire history is stored.

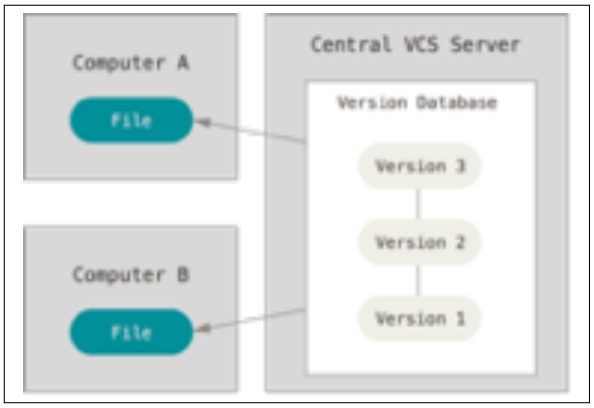


Figure 2.1: Structure of a CVCS. Source: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control, accessed on June 21, 2022

[7]Instead of saying "Distributed Version Control System", some sources refer to such systems as "Decentralized Version Control Systems", see for example de Alwis and Sillito (2009). In this thesis, we use both terms interchangeably.

[8]http://cvs.nongnu.org, accessed on June 21, 2022

[9]https://subversion.apache.org, accessed on June 21, 2022

Distributed Version Control Systems

In a DVCS such as Git[10], there is no central repository. Every user has a full copy of a given repository on their computer. Changes are always performed on this local repository. So, a user can directly create commits, branches, tags, or the like, on the locally stored repository. To enable collaboration with other users, it is possible to either make the locally stored repository available over a network, or to publish this repository on a server. Other users can then *clone* this repository to their own computers and work independently of the original repository. When collaborating with other users, one *pushes* the locally made changes to the repository the clone has been made from. If another user has pushed a change, it is possible to *fetch* the changes to the locally stored repository. When performing a fetch operation, it is necessary to merge the incoming changes with the ones already present. (Otte, 2009)

Figure 2.2 visualizes how a DVCS is organized: The entire history is always stored on every single computer. It is possible to exchange information with any other computer, which is the major difference to the client-server model used in CVCSs.

Figure 2.2: Structure of a DVCS. Source: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control, accessed on June 21, 2022

[10]https://git-scm.com, accessed on June 21, 2022

2.3 Trees and graphs

Two data structures are essential for the VCS implemented in this thesis: trees and graphs. The following explanations are based on the definitions of Mehlhorn and Sanders (2008).

2.3.1 Trees

A tree is a data structure used in computer science and similar fields to model hierarchies between entities. These entities are represented by nodes in the tree. Every node can have multiple "children". Additionally, every node has exactly one "parent" node, except for one node, which is called the "root". The terms "child" and "parent" are inverse to each other: Given two nodes A and B, if B is a child of A, then A is the parent of B. Finally, in this thesis we call nodes without children "leaves", and all other nodes "inner nodes".
Figure 2.3 shows an example of a tree with five nodes. Node 1 is the root node and has two children (nodes 2 and 3). Thus, node 1 is the parent of nodes 2 and 3. Nodes 4 and 5 are children of node 3. Nodes 2, 4 and 5 are leaf nodes. In this example, a node has a maximum of two children. However, if no further restrictions are defined, a node may have an arbitrary number of children.

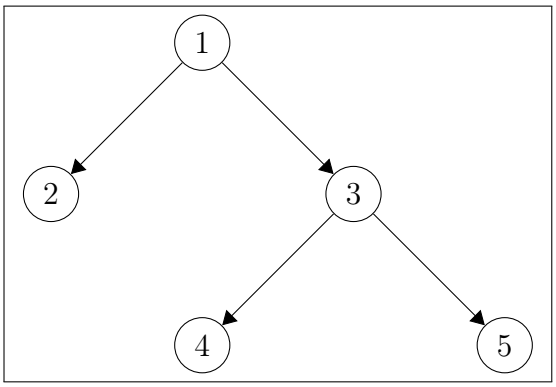


Figure 2.3: An example of a tree with five nodes (nodes 1 to 5). Nodes 2 and 3 are children of node 1; nodes 4 and 5 are children of node 3.

Application in file systems

One situation where tree(-like) structures can be found are file/folder structures in a typical file system: Here, a folder can contain multiple sub-folders and files. Every file and folder (except for the root folder) has exactly one parent folder. This is exactly what trees are able to represent: When mapping the file system

model to a tree, every single file and folder is represented by one node. A file node never has any children (it only holds the content of the respective file it represents, but that can be seen as being "part of the node"). A folder node references the file and folder nodes that represent the direct children of the given folder. Thus, every file node is always a leaf node, while folder nodes are either inner nodes (if the folder is not empty), or also leaf nodes (if the folder is empty).

Hash trees

A special type of tree is the hash tree, which is based on a technique described by Merkle (1987, 1989a). Assuming that all leaf nodes are meant to reference some kind of payload, every leaf node also stores the hash[11] of its respective payload. Inner nodes, on the other hand, store a hash based on the hashes of their child nodes.

Hereinafter, we use the following definitions[12]: Let N_x be a node in a hash tree, with x being its unique identifier ("label"). $C(N_x)$ shall be the sequence of child nodes of N_x. (We say "sequence" instead of "set", since we want to assume some kind of stable ordering of the child nodes.) For convenience, we say that $C_i(N_x)$ is the i-th child node of N_x. Let H_x be the hash of node x, and hf be the hash function used for the computation of all hashes in this hash tree. We say that every leaf node l holds data (a payload) D_l.

For every leaf node N_l, the hash H_l is defined as the hash of its payload: $H_l = hf(D_l)$. For every inner node N_i with $n = |C(N_i)|$ children, the hash is defined as the hash of the concatenation of the hashes of its children: $H_i = hf(H_{c1} \oplus H_{c2} \oplus ... \oplus H_{cn})$. Here, we say that the operator $\oplus$ is a concatenation, and that $c1$, $c2$, and so on, are the identifiers of the children of node N_i. This shows an interesting aspect of hash trees: The hash of a node "covers" all descendants (children, children of children, etc.) of this node.

[11] A hash function, as shown by Merkle (1989b), is a function that computes a "fingerprint" or "cryptographically secure checksum[]" of a given piece of information.

[12] The definitions are based on the explanations by Niaz and Saake (2015) and Szydlo (2004), but have been generalized to trees consisting of nodes with arbitrary numbers of children.

Hence, if two nodes have the same hash, then this means that all descendants of this node also have the same hash and are thus identical. This is particularly useful when comparing two trees, e.g. in order to determine which nodes are different between these two trees: Without hashes, in order to determine which nodes are different (e.g. that have different children or, in the case of a leaf node, a different payload), it is necessary to recursively compare every single node. A simple "diff" algorithm could be defined as follows:

$$Diff(N_a, N_b) = \begin{cases} (N_a, N_b) \cup \bigcup_{n=1}^{max(|C(N_a)|,|C(N_b)|)} Diff(C_n(N_a), C_n(N_b)) & \text{if } N_a \neq N_b \\ \bigcup_{n=1}^{max(|C(N_a)|,|C(N_b)|)} Diff(C_n(N_a), C_n(N_b)) & \text{otherwise} \end{cases}$$

Let $T1$ and $T2$ be two trees that shall be compared. Let N_{T1} be the root node of $T1$, and N_{T2} be the root node of $T2$. To compare these two trees without the usage of hashes, it is necessary to compute $Diff(N_{T1}, N_{T2})$, which recursively compares **all** nodes of the two trees[13]. However, in a hash tree, it is possible to utilize the hash of a node to determine if the comparison of two subtrees can be skipped:

$$HashDiff(N_a, N_b) =$$

$$= \begin{cases} (N_a, N_b) \cup \bigcup_{n=1}^{max(|C(N_a)|,|C(N_b)|)} HashDiff(C_n(N_a), C_n(N_b)) & \text{if } H_a \neq H_b \\ \emptyset & \text{otherwise} \end{cases}$$

This definition shows that the usage of a hash allows to skip the comparison of subtrees if their hash is the same, since this indicates that the subtrees themselves are recursively equal.

It should be noted that in this simple definition of $Diff(N_a, N_b)$ (and also of $HashDiff(N_a, N_b)$), we compare the nodes in order, i.e. we compare child node $C_1(N_a)$ with child node $C_1(N_b)$, then we compare child node $C_2(N_a)$ with child node $C_2(N_b)$, and so on. This is shown in the union

[13]It should be noted that in the definition of $Diff(N_a, N_b)$, we use the comparison $N_a \neq N_b$. We say that this compares the payloads if two leaf nodes are given, i.e. it is *true* if the two payloads are unequal. If two inner nodes are given, this comparison "waits" for the result of the recursive comparison of the descendant nodes and then checks if the result is non-empty. In all other cases not mentioned here, $N_a \neq N_b$ is defined to be *true*.

$\bigcup_{n=1}^{max(|C(N_a)|,|C(N_b)|)} Diff(C_n(N_a), C_n(N_b))$. Technically, a "smarter" selection of node pairs could improve the results of these comparisons, but they especially depend on the concrete use case.

Figure 2.4 shows an example of a hash tree. This tree is structured in the same way as the tree shown in figure 2.3. Every node is labelled with its hash: Nodes 2, 4 and 5 are leaf nodes, thus the hash is calculated using their respective payload (D_2, D_4, and D_5). Nodes 1 and 3 are inner nodes. Because of this, their hash is calculated using the hash of their respective children. This figure also shows that hashes need to be computed "bottom-up", i.e. starting at the leaf nodes: In order to compute the hash of node 1, it is necessary to compute the hashes of nodes 2 and 3. And to compute the hash of node 3, it is necessary to compute the hashes of nodes 4 and 5.

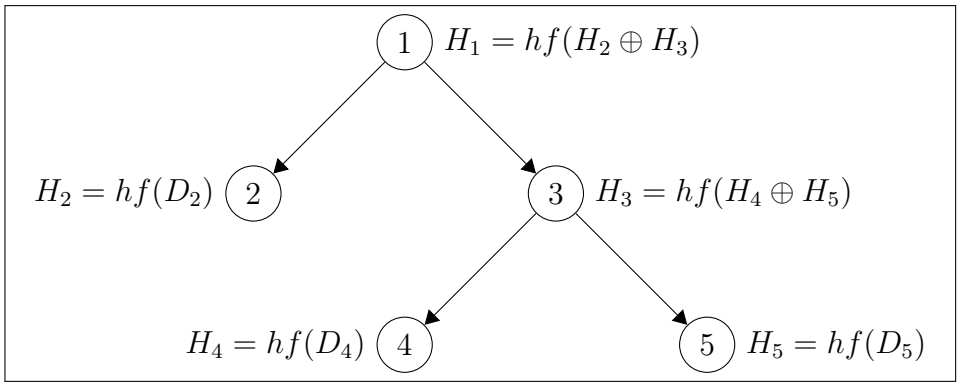


Figure 2.4: An example of a hash tree, based on the tree shown in figure 2.3.

When applying hash trees to the model of file/folder trees in file systems, it allows to quickly determine if two folders are equal by comparing their hashes: If two folders have the same hash, then all sub-folders (and their sub-folders and so on), as well as all files in these folders, are equal. This makes hash trees particularly useful when files and folders frequently need to be compared: In order to (recursively) determine the differences between two folders, it is only necessary to further compare those children that have a different hash.

2.3.2 Graphs

The concept of graphs forms a superset of the concept of trees: Similar to a tree, a graph allows to model the relationship between entities (represented by nodes). However, these entities do not have to form a hierarchy. Instead, two arbitrary nodes can be connected using an edge to indicate that some kind of "connection" exists between them. An edge can be directed, which is used to

indicate a direction of the respective connection. A graph is acyclic if, when traversing the graph, there is no path where one node appears more than once. A graph is called a Directed Acyclic Graph (DAG) if all edges in this graph are directed and the graph is acyclic.

Figure 2.5 shows an example of a DAG: Here, five nodes are connected with multiple directed edges. When traversing this graph following the "direction" of the edges, it is not possible to visit a particular node twice, which makes this graph acyclic.

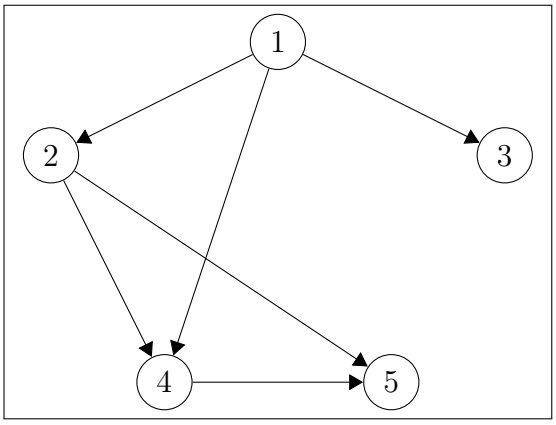


Figure 2.5: An example of a DAG with five nodes (nodes 1 to 5).

3 Requirements

This chapter explains functional and non-functional requirements that shall be fulfilled by the VCS built in the context of this thesis.

3.1 Source of requirements

The requirements are split into functional and non-functional requirements. The functional requirements are based on the Core Operations as described in chapter 2; the non-functional requirements are defined based on the context in which the Hub VCS will be developed.

3.1.1 Adaptation of the Core Operations

As already mentioned, the functional requirements are based on the Core Operations. However, since Sink (2011) defines certain operations like *diff* in a different way compared to how it is done in section 2.2.1, it was necessary either to add additional functional requirements or to slightly diverge from the idea behind certain Core Operations.

Additionally, the decision was made to skip two Core Operations: *Core Operation 5 (Add)* and *Core Operation 18 (Lock)*.

Core Operation 5 (Add) implies that it is possible to create untracked files in the working copy (and later to be able to advise the VCS to explicitly track these files). Such a feature would not provide any advantages when used in the context of the DEWB, since the only reason for a user to create a file in the DEWB is because he or she wants to publish a configuration on the JValue Hub. Thus, there will be no functional requirement based on *Core Operation 5 (Add)*.

Core Operation 18 (Lock) essentially refers to an exclusive locking mechanism allowing one user to prevent other users from modifying one or multiple given files. Allowing users to stop other users from working on the same files could have

a major negative impact on the user experience in the DEWB. Such a feature would mean that one user could stop other users from working on a particular project in the DEWB for a potentially long period of time. Such a feature can be replaced by implementing proper merge features. Because of this, *Core Operation 18 (Lock)* will not be part of the functional requirements.

3.2 Requirements format

The functional requirements are defined in the format of User Stories. User Stories are often used in agile development and commonly follow the format "As a ⟨type of user⟩, I want ⟨goal⟩, [so that ⟨some reason⟩]" (Lucassen, Dalpiaz, van der Werf & Brinkkemper, 2016). Thus, a User Story defines which stakeholder a feature is meant for, what the actual feature consists of (e.g. which action the stakeholder wants to be able to perform), and (optionally) includes a clarification or other type of information.

In this thesis, we allow functional requirements to contain more than one User Story. This helps to avoid an overly large number of requirements. Also, we allow User Stories to slightly violate the strict sentence format mentioned before in order to make them easier to comprehend.

User Stories are meant to be "problem-oriented" (Lucassen et al., 2016) and should not include technical details. Thus, the functional requirements formulated in this thesis do not define how or "where" a particular feature shall be implemented: It is for instance possible that a feature only exists in the client part (frontend) of the system and internally utilizes other features provided by the server part (backend). As an example, it could be possible that the backend has a feature to fetch all files for a given commit, but that no diff feature exists on the backend. To display a diff between two commits, the frontend could load the folder trees of the two commits and calculate the diff on the client side. The decision how to approach a certain feature is not defined in the functional requirements, but must be selected when actually implementing this feature.

At the end of this thesis, the features of the final VCS will be compared against the list of functional requirements, in order to evaluate by which degree the requirements have been met. User Stories are very well suited for this purpose, because every User Story can be evaluated independently of the others.

3.3 Metric to summarize requirements completion

In order to summarize by which degree the functional and non-functional requirements are met by the final system, the evaluation presented at the end of this thesis will contain a metric that shows the fraction of successfully completed requirements. For this purpose, every single requirement will be classified as "completed" or "failed". Even if only a small part of a requirement has not been fulfilled, it will be classified as failed (i.e. there is no "partially failed" or the like). Then, both for the functional and for the non-functional requirements, the fraction $\frac{\#\text{CompletedRequirements}}{\#\text{TotalRequirements}}$ will be computed. #CompletedRequirements is the number of completed requirements, and #TotalRequirements is the total number of requirements. This fraction will be computed separately for the functional and for the non-functional requirements, which means that in the end, there will be one number that shows by which degree the functional requirements have been met, and another number that shows by which degree the non-functional requirements have been met. No weighting of individual requirements takes place, which means that all requirements are treated as equally important in this metric.

3.4 Functional requirements

In order to be able to reference individual functional requirements, these requirements are numbered and prefixed with "F-".

F-1: **Repository Listing**

As a user of the Hub VCS, I want to retrieve a list of all existing repositories, so that I can get an overview of all existing repositories and eventually start exploring one in more detail.

F-2: **Repository Creation**

As a content creator, I want to create new repositories, so that I can share my work with other users.

This requirement is based on *Core Operation 1 (Create)*.

F-3: **Repository Deletion**

As a repository owner, I want to delete my repositories, so that I can remove content I don't want to share any longer.

F-4: **Branch Listing**

As a user accessing a particular repository, I want to get a list of all existing branches, so that I can explore the work done in this repository.

This requirement is based on *Core Operation 15 (Branch)*.

F-5: **Branch Management**

As a collaborator in a repository, I want to create new branches, so that I can work independently of other users.

Additionally, I want to be able to rename existing branches, so that I can fix potential spelling mistakes or improve the understandability of a name.

Finally, I want to be able to delete branches, so that I can remove content that is not needed anymore.

This requirement is based on *Core Operation 15 (Branch)*.

F-6: **Tag Management**

As a collaborator in a repository, I want to create new tags, so that I can highlight important commits.

Additionally, I want to edit existing tags, so that I can improve already existing tags instead of having to create new ones.

Finally, I want to delete tags, so that I can remove tags that are not needed anymore.

This requirement is based on *Core Operation 14 (Tag)*.

F-7: **Checkout**

As a collaborator in a repository, I want to checkout a particular revision (commit), so that I can start performing changes to it.

This requirement is based on *Core Operation 2 (Checkout)*.

F-8: **Edit**

As a collaborator in a repository who has checked out a revision, I want to edit the content of the files in my working copy, so that I can try out and apply the changes I had in mind.

This requirement is based on *Core Operation 6 (Edit)*.

F-9: **File and Folder Management**

As a collaborator in a repository who has checked out a revision, I want to create new files and folders in my working copy, so that I can add new content to the repository.

Also, I want to delete files and folders, so that I can remove content I don't need anymore.

Additionally, I want to rename and move files and folders, so that I can easily change the directory structure according to my ideas.

This requirement is based on *Core Operation 7 (Delete)*, *Core Operation 8 (Rename)* and *Core Operation 9 (Move)*.

F-10: **Status and Revert**

As a collaborator in a repository who has checked out a revision and performed changes to the working copy, I want to see the changes I have performed compared to the originally checked out version, so that I can get a summary of what I have done.

I don't only want to see which files have been changed, but also get a more in-depth comparison of the changed files, so that I can understand the changes I have performed in detail.

Additionally, I want to be able to revert changes I have made to my working copy, so that I can undo modifications that aren't needed anymore.

This requirement is based on *Core Operation 10 (Status)*, *Core Operation 11 (Diff)* and *Core Operation 12 (Revert)*. It should be noted that *Core Operation 11 (Diff)* only refers to the diff between the originally checked out version and the current changes in the working copy. It does not refer to features related to the comparison of two commits or the like.

F-11: **Commit**

As a collaborator in a repository who has checked out a revision and performed changes to the working copy, I want to commit the changes I have made, so that I can share them with other users.

This requirement is based on *Core Operation 3 (Commit)*.

F-12: **Update**

As a collaborator in a repository who has checked out a revision, I want to update my working copy if it does not point to the head of the checked-out branch anymore, so that I can base my work on the latest revision.

If I have already performed changes to my working copy, I expect the incoming changes to be merged with the ones I have performed, so that I don't have to start over again.

This requirement is based on *Core Operation 4 (Update)*.

F-13: **Merge**

As a collaborator in a repository, I want to merge one branch onto another one, so that I can combine the changes that have been developed individually on those branches without having to manually copy these changes.

If the merge cannot be performed automatically, I want to be able to resolve the conflicts manually.

This requirement is based on *Core Operation 16 (Merge)* and *Core Operation 17 (Resolve)*.

F-14: **History**

As a user accessing a particular repository, I want to see the history of the repository (i.e. the list of commits), so that I can retrace the historic development of this repository.

Additionally, I want to see the history of individual files (i.e. see which commits affected those files), so that I can find out when and why a particular file has been modified.

This requirement is based on *Core Operation 13 (Log)*.

F-15: **Diff**

As a user accessing a particular repository, I want to see the diff between two commits, so that I can understand the detailed changes that have been performed between these two commits.

3.5 Non-functional requirements

In order to be able to reference individual non-functional requirements, these requirements are numbered and prefixed with "N-".

N-1: **Architecture**

Because the DEWB will become a fully web-based editor, the backend of the Hub VCS should be accessible using an HTTP API.

The frontend of the Hub VCS should also be web-based so that it will later be possible to integrate the frontend of the Hub VCS into the DEWB (or vice versa).

N-2: **Programming Languages and Frameworks**

The backend and frontend should be built using TypeScript[1]. Using the same programming language across multiple parts of the system might (and most likely will) allow to reuse pieces of code. Since the DEWB and other projects in the context of the JValue Project are also mostly developed using TypeScript, it is expected to achieve the best interoperability when also developing the Hub VCS using TypeScript.

The frontend should additionally be developed using Vue.js[2], since the DEWB and other projects in the context of the JValue Project are also based on this library.

N-3: **Code Style**

The code style should match the style used in other software applications developed in the JValue Project.

N-4: **Testing**

All HTTP API endpoints should be covered by automated system tests. At least one test case should be created for every single API endpoint.

N-5: **API Documentation**

There should be documentation for every single HTTP API endpoint, describing which features it provides, which payload a request should contain, and which data the response will contain. Since TypeScript is used both on the server and the client(s), the request payload and response data may be documented in code using TypeScript interfaces or similar.

[1] https://www.typescriptlang.org, accessed on June 21, 2022
[2] https://vuejs.org, accessed on June 21, 2022

4 Conceptual design

In this chapter, important theoretical concepts and decisions are explained that form the foundation for the implementation of the Hub VCS. These concepts are independent of the concrete implementation of the Hub VCS and could therefore also be used as a foundation for the implementation of other VCSs that are meant to provide similar features as the Hub VCS.

4.1 Comparison of CVCSs and DVCSs

A very important architectural decision is to determine whether a CVCS or a DVCS shall be developed. As described in chapter 2, the major difference between these two architectures is that a CVCS is built based on a client-server-like architecture, while a DVCS makes use of a distributed workflow where every repository can essentially exist independently of any other repository.

Another way to compare CVCSs and DVCSs is to look at how the content of a repository is accessed. In a CVCS, the clients access the current revision by performing a checkout for one specific revision. In a DVCS, clients first need to create a copy of the repository (often including the entire history) and can then access all revisions without even having to rely on an internet connection.

To answer the question which of these two architectural styles might be better suited, it is important to take the context of development into account. Most importantly, the features and requirements of the DEWB need to be complied with, since the VCS will be tightly connected to the DEWB.

The DEWB is meant to be a completely web-based editor. Since one of the most important features of it relies in the ability to preview data from remote sources, it is also built with a permanent internet connection in mind. Thus, it is also not necessary for the VCS to be usable in an offline scenario. Additionally, the JValue Hub is meant to be the single point where configurations are permanently stored. There is no need to allow clients to work independently of the JValue Hub,

especially since the DEWB will rely on APIs provided by a service connected to the JValue Hub.

Because of these reasons, the decision was made to implement a CVCS instead of a DVCS. As already defined in requirement N-1, the VCS will be required to provide an HTTP API and to have a web-based frontend.

4.2 Reusing existing general-purpose systems

Before being able to implement the Hub VCS, the decision must be made whether the system shall be implemented from ground up ("from scratch"), or if it could be based on an already existing (general-purpose) VCS such as Git or SVN.

There are already software packages like GitLab or Gitea[1] that enable administrators to self-host a full VCS. These packages provide both a backend including an HTTP API, as well as a frontend allowing the users to interact with repositories and the like. In the following, we call such systems "self-hosted VCSs".

4.2.1 Reusing user interfaces

Unlike the general-purpose approach of self-hosted VCSs, the JValue Hub will be specifically tailored to the needs of data scientists and developers working on configurations of the ODS. This especially brings many requirements regarding the user interface (frontend) that are not supported by the user interfaces of the aforementioned self-hosted VCSs. One option to solve this issue would be to extend an already existing frontend with the features required by the JValue Hub. However, since the requirements of the JValue Hub will lead to many large additional features with deep integration into many parts of the user interface, the decision was made to rather implement a completely new frontend.

4.2.2 Reusing HTTP APIs

Since many self-hosted VCSs also provide an HTTP API, it could be seen as a good approach simply to use such an API when implementing the custom frontend. Alternatively, a custom HTTP API could be implemented based on features directly provided by a system like Git or SVN.

In the short term, compared to a completely new implementation, both approaches would most certainly require much less time and effort for the develop-

[1] https://gitea.io, accessed on June 21, 2022

ment of a system fulfilling the minimal requirements. Also, since these systems have already been used by many other projects, it is very likely that most features have already been well tested.

However, as mentioned earlier, the JValue Hub has special requirements regarding the features of the VCS. In the future, it could for instance be very beneficial for the user experience if a diff between two commits is also able to take semantic information into account, instead of just comparing files line-by-line. Additionally, partitioning a repository into multiple "feature repositories" could be required if, for instance, not only data source configurations, but also other types of information like data coming from remote APIs shall be versioned. Furthermore, fine-grained access permissions could add a layer of security for collaboration in larger repositories. Finally, it might be required to attach additional meta data to commits, such as flags telling the JValue Hub whether an API described by a data source configuration should be queried.
Adding such complex additional features to the data model of Git or SVN would require large changes to the source code of these systems. Since the VCS is such a critical component regarding the success of the JValue Hub, the decision was made to implement the entire VCS from scratch. In the long run, this especially enables the implementation of unique features for the JValue Hub, without being limited by a data model that is not specifically suited for the requirements of data scientists and the like.

4.3 Logical Model of the Hub VCS

The central and probably most critical part of a VCS is its data model: Without a good model of repositories, branches, commits, and the like, a VCS will be of little use. Thus, it is critical to determine which types of information need to be stored by the VCS, and how entities are related to each other. In this thesis, we call this the "Logical Model".

4.3.1 Initial Logical Model

Section 3.4 describes the features the final system should provide. This set of features also indirectly contains a high-level overview of the pieces of data that need to be provided by the VCS, which is a basis for the Logical Model:

- Requirement F-1 says that it should be possible to retrieve a list of all existing repositories. This implies that the VCS should be capable of storing multiple repositories.

- Requirement F-3 mentions that repository owners shall be able to delete a repository. This implies that some kind of repository ownership exists, which in turn requires user accounts to exist. Since no further restrictions were made, a simple solution can be selected at this point: One repository has exactly one owner (which is a user). One user may own multiple repositories. This is a **1:N** (One-to-N) relationship between users and repositories.
- Requirement F-7 says that it shall be possible to checkout a particular revision (i.e. a commit). This implies that revisions (commits) need to be stored in the system. The requirement does not explicitly state that a commit should be part of a repository. However, this is indirectly introduced by F-14, because this requirement says that the history of a repository is equal to the list of commits. Thus, we say that a commit always belongs to a repository, and that a repository can contain multiple commits, which is a **1:N**[2] relationship between repositories and commits.
- Requirement F-11 talks about the creation of commits and that changes made in the working copy are part of a commit. This implies that it is possible to store files and folders, and/or some kind of "delta" (i.e. diff to the previous commit), in the VCS. Also, commits and files/folders/delta are connected in some way. At this point, it is not clear how this part of the architecture shall look like. For instance, it could be necessary to store both the delta and a "snapshot" of the files/folders for a commit.
- Requirement F-14 explains that it shall be possible to retrieve the commit history of a repository, and that it shall be possible to retrieve the history of a particular file (i.e. the list of commits that modified a particular file). At this point, it is not clear how such a data model could look like. Most likely, it depends on how commits and their files/folders/delta are represented.
- Requirement F-4 implies that a particular repository should contain a list of branches. Thus, one repository shall be able to contain N branches. This is a **1:N** relationship between repositories and branches, since the requirements do not mention that it shall be possible for one branch to belong to multiple repositories.
  Additionally, a branch references a commit, and a commit can be referenced by multiple branches, which leads to a **1:N** relationship between commits and branches.
- Similar to how a repository shall contain branches, requirement F-6 describes that a repository should contain a list of tags. This is also a **1:N**

[2]Every usage of "N" shall imply that a "fresh" "N" is being used. For instance, the number of commits of course does not have to be equal to the number of repositories. "N" is merely a way to express cardinality here.

relationship between repositories and tags, since the requirements do not mention that tags should be shared across the "borders" of repositories. Additionally, a tag references a commit, and a commit can be referenced by multiple tags, which leads to a **1:N** relationship between commits and tags.

Figure 4.1 shows a graphical representation of the Logical Model. The figure also shows that two aspects of the Logical Model are still missing:

1. It is not clear how files, folders, and the delta for a commit shall be modelled.
2. It is also not clear how a model for the commit history could look like.

Both of the missing parts of the Logical Model will be discussed hereinafter.

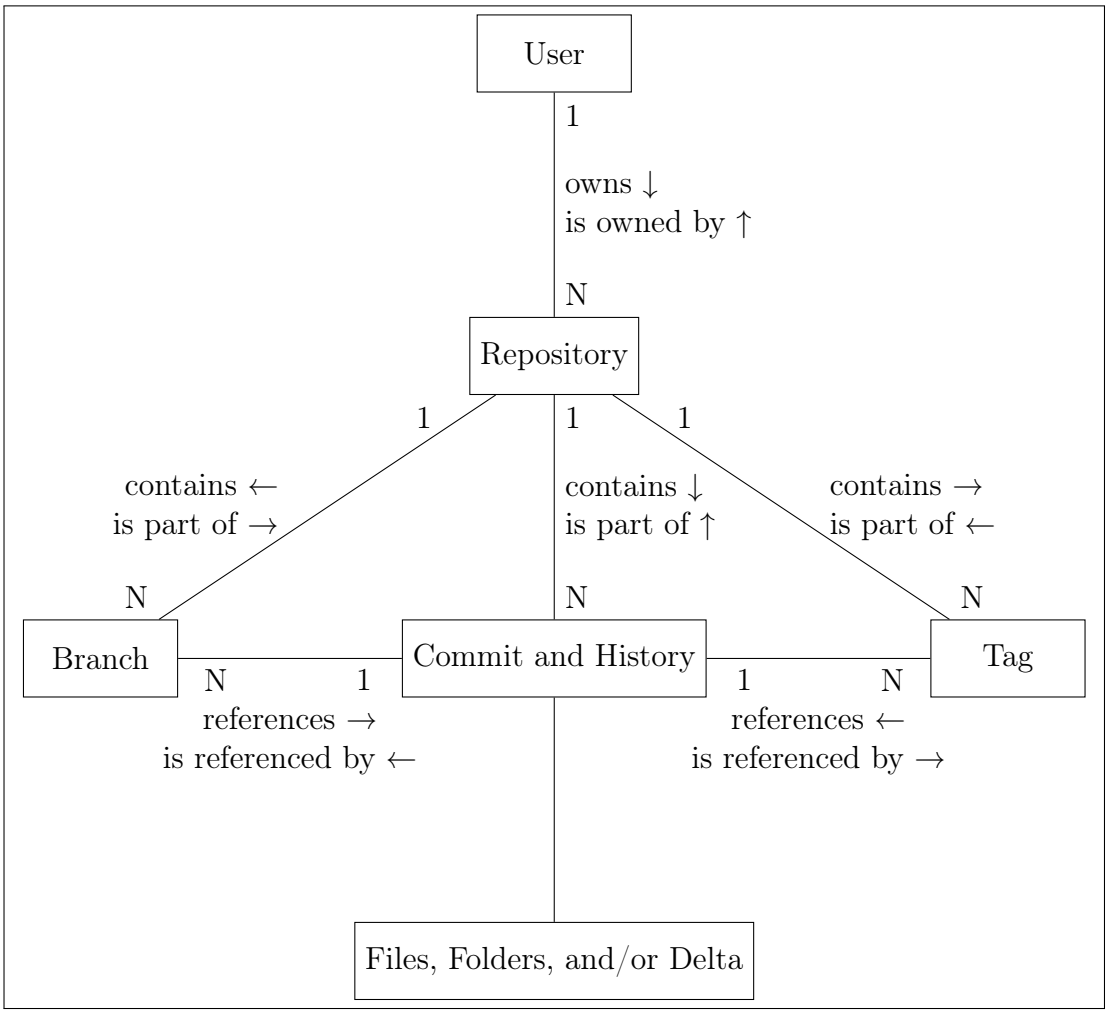


Figure 4.1: Overview of the Logical Model. It is not clear how the commits and their history shall be represented, and how they are related to files, folders, and a delta to the previous version.

4.3.2 Model of files, folders, and delta

In order to make it possible for the client to actually perform a checkout on a particular commit, it must be possible to fetch the state of the files and folders at the point a particular commit was made. Additionally, it should be possible to compare the files and folders of two commits in order to allow the client to visualize a diff.

A simple model for this purpose (in the following called "Simple Storage" model) could be designed as follows: Every commit references a (root) folder. Every folder can contain multiple sub-folders, and multiple files - similar to a file system. This is visualized in figure 4.2. When a new commit is created, then a copy of all folders and files (in the following called "tree items") is made before applying the delta (i.e. the changes) to this copy. Figure 4.3 shows an example where one commit ("Commit 1") references a folder with two files ("File A" and "File B"). Based on this commit, a new commit ("Commit 2") is created. In this new commit, a new file is introduced ("File C"). The other files remain unchanged.
The advantage of this simple model is that it is always possible to retrieve all files and folders for a checkout without having to rely on possibly expensive computations: Every commit always points to "its own" file/folder tree, making it possible to access the file/folder tree in constant time ($\mathcal{O}(1)$). However, this also means that if a commit only changes a small subset of the tree items compared to its predecessor, a large number of unchanged tree items are copied to be referenced by the new commit. Thus, the number of tree items stored in the VCS grows linearly with the number of commits, with a potentially large number of duplicates - even small changes require a full copy of all tree items.

There are two models that avoid the issue of having to copy (a potentially large number of) unchanged tree items as described by Chacon and Straub (2022):

Systems like CVS or SVN only store the changes (deltas) that are introduced by a new commit. In the following, this mechanism is referred to as "Delta Storage". On the other hand, Git stores snapshots similar to the Simple Storage model described before, but uses references to avoid duplicate tree items. In the following, this mechanism is referred to as "Snapshot Reference Storage".[3]

Delta Storage

As described before, Delta Storage avoids the problem of having to copy all tree items for every commit by only storing the deltas compared to other revisions. An example of this mechanism is illustrated in figure 4.4: Here, three files ("File

[3]The terms "Delta Storage" and "Snapshot Reference Storage" are no "official" terms. Instead, they have been introduced in this thesis to avoid confusion with other terms.

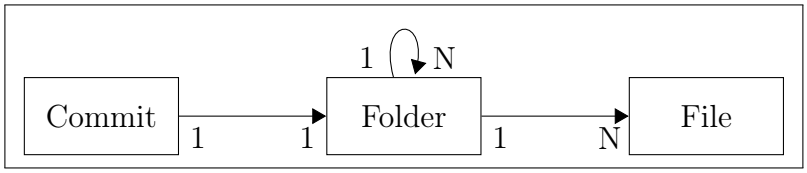


Figure 4.2: The Simple Storage model: A commit points to a (root) folder. A folder can contain multiple sub-folders, and also multiple files. This is similar to how a file system is organized.

A", "File B", and "File C") are being tracked in a VCS. The "base" of these files can be found at Version (Revision) 1. In Version 2, files A and C get modified. This is done by storing the differences (deltas) compared to the respective state at Version 1. In Version 3, File C gets modified again. This time, the delta compared to Version 2 is stored. Versions 4 and 5 are handled in a similar way.

A problem with this storage mechanism is the computational complexity for retrieving the file/folder tree of one particular revision. In the following, to simplify the explanation, the assumption is made that only one single file is tracked by the system. Over time, this file gets modified multiple times, resulting in multiple revisions (versions). When using a mechanism as shown in figure 4.4, the retrieval of the file content in version 1 is trivial. In order to retrieve the file content in version 2, it is necessary to apply the delta found at version 2 (in the following called Δ_2) to the file content found at version 1. In order to retrieve the file content at version t, it is necessary to apply all deltas Δ_2 to Δ_t to the file content found at version 1. (Koc & Tansel, 2011) This means that the number of deltas that need to be applied in order to retrieve the file content of a particular revision is in $\mathcal{O}(t)$, which could lead to problems if many commits shall be handled by the system.
To address this issue, some VCSs use more advanced algorithms with a sub-linear complexity. For instance, according to the SVN Developer Notes[4], and Vaidya, Torres-Arias, Curtmola and Cappos (2019), SVN uses a mechanism called "skip-deltas" when computing and storing the deltas for a new commit. This mechanism reduces the number of deltas that need to be applied in the aforementioned scenario to $\mathcal{O}(lg(t))$. Furthermore, the SVN Developer Notes, and Koc and Tansel (2011), describe a "reverse-delta" mechanism where the deltas are computed relative to the latest revision instead of the first revision. This is useful in scenarios where the latest commits get accessed more often than the older ones. However, if there are n commits stored in a system that is using a combination of the skip-deltas and reverse-deltas algorithms, it is still on average necessary to apply $\mathcal{O}(lg(n))$ deltas to retrieve the file content of a particular revision.

[4]See https://svn.apache.org/repos/asf/subversion/tags/1.9.11/notes/skip-deltas, accessed on June 21, 2022

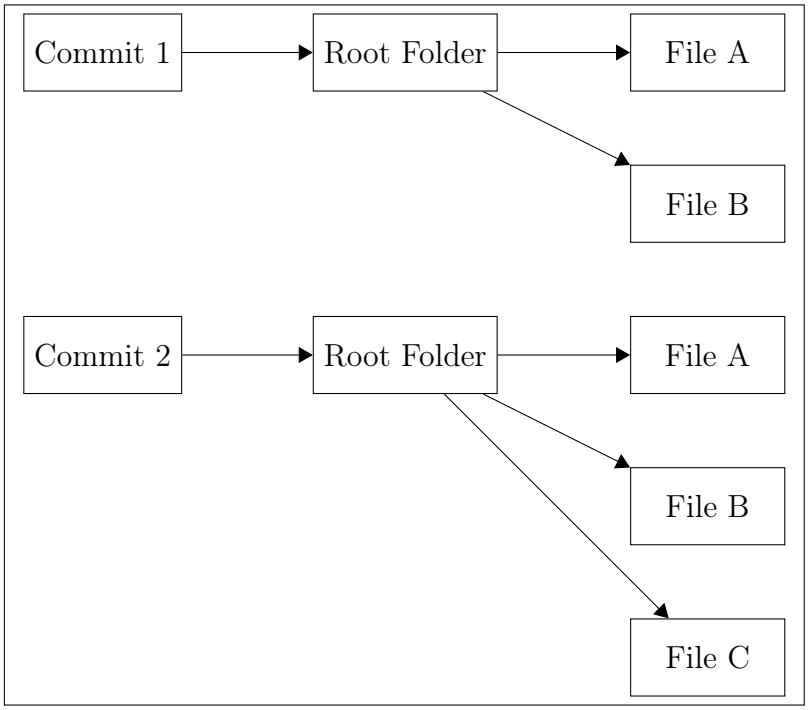


Figure 4.3: Two commits in the Simple Storage model. Commit 1 references a folder ("Root Folder") with two files ("File A" and "File B"). In commit 2, a new file ("File C") has been introduced. For the creation of commit 2, files A and B (and the Root Folder) have been copied.

Snapshot Reference Storage

Snapshot Reference Storage is a mechanism that sets Git apart from many other VCSs. Instead of storing deltas, Snapshot Reference Storage actually stores full file/folder trees, similar to the Simple Storage model. But instead of copying unchanged tree items, Git simply creates references to all unchanged tree items in order to avoid the necessity to store duplicate content. (Chacon & Straub, 2022) This mechanism is shown in figure 4.5. This figure shows the same series of changes as figure 4.4, but this time using the Snapshot Reference Storage model. In Version 1, there are again three files ("File A", "File B", and "File C") that are being tracked by the VCS. In Version 2, files A and C get modified. The new file versions are directly stored ("A1" and "C1"), instead of just storing a delta. File B remains unchanged, but can still be directly accessed when fetching Version 2, because a reference (pointer) is used to reference the actual file found at Version 1. In figure 4.5, this is represented using a dotted border. In Version 3, File C is modified again and is now stored as "C2". The other two files remain unchanged and are thus represented as references to files A1 and B. Versions 4 and 5 are handled in a similar way.

Applied to the example in Figure 4.3, instead of copying files A and B, the Snapshot Reference Storage model would use references. Thus, in commit 2, files A and B would be represented by references to the files found in commit 1, as shown in figure 4.6.

The Snapshot Reference Storage model brings an important advantage compared to the Delta Storage model: The file/folder tree for every commit can be directly accessed, without having to apply multiple deltas to a base version. This makes the Snapshot Reference Storage model effectively independent of the number of commits: Accessing the file/folder tree for commit n does not depend on any predecessors or successors of commit n. Thus, it is possible to access the file/folder tree in constant time ($\mathcal{O}(1)$). Compared to the Simple Storage model, the Snapshot Reference Storage model has the advantage of not storing potentially large numbers of copies of tree items.

The Snapshot Reference Storage model combines the constant access times found in the Simple Storage model with the ability to avoid large numbers of duplicate tree items as described in the Delta Storage model. Because of these advantages, the Snapshot Reference Storage model has been selected as the model to be used for the Hub VCS.

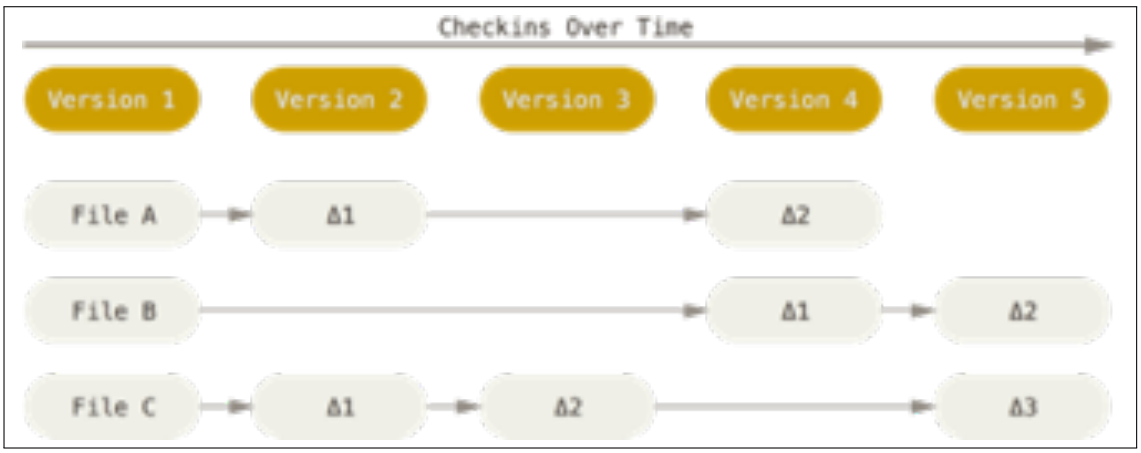


Figure 4.4: Delta Storage as a mechanism to avoid full copies of all tree items for every commit. Source: Chacon and Straub (2022)

4.3.3 Model of the commit history

The last missing aspect of the Logical Model is related to the Commit History. More precisely, it is not clear how the Commit History itself shall be modelled and how it interacts with the Storage Model described in the previous section.

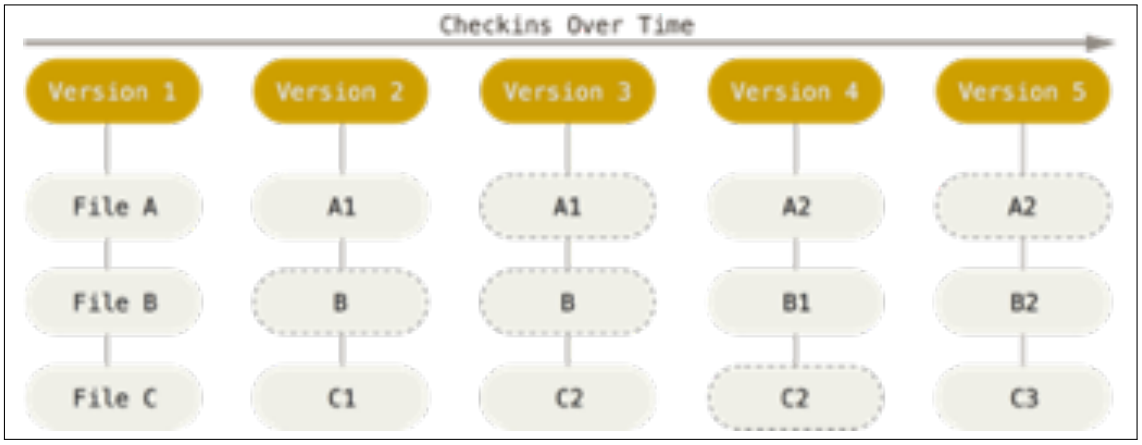


Figure 4.5: Snapshot Reference Storage as a mechanism to avoid full copies of all tree items for every commit. Source: Chacon and Straub (2022)

Sink (2011) describes two history models: The Linear Model, and the DAG Model.

The Linear Model describes the commit history as one "line" of commits: Every commit has exactly one direct predecessor, and exactly one direct successor. The only exceptions are the very first commit, which has no predecessor, and the latest commit, which has no successor. In the following, the direct predecessor of a commit is referred to as "commit parent".
Figure 4.7 shows a linear commit history: In this figure, a commit is represented by a circle, and parents are indicated using an arrow pointing from a particular commit to its parent. The revision numbers of every commit are represented by the numbers within the circles. Every commit, except for the very first one, has exactly one parent, and every parent has a lower revision number compared to its successor.

In order to create a new commit, the client fetches the latest commit C_n, performs changes to it, and then creates a new commit C_{n+1}. C_{n+1} is now the latest commit, with C_n being its parent. Figure 4.8 shows the commit history after commit C_{n+1} has been created, based on the commit history shown in figure 4.7.

This history model unfortunately does not solve one problem: It is not clear how merges between branches should be handled. Figure 4.9 shows an example scenario: Two users "A" and "B" are working on the same project. Commits C_1 to C_{n-1} are part of branch "A". User A checks out commit C_{n-1} and directly makes changes on the same branch, resulting in commits C_n and C_{n+1}. User B also wants to perform some changes, but in order not to interrupt the workflow of user A, he or she creates a separate branch, branch "B". This branch is based on commit C_{n-1}. Now, user B creates two commits on this branch: C_b and C_{b+1}. This can be done completely independently of user A, since the two users are

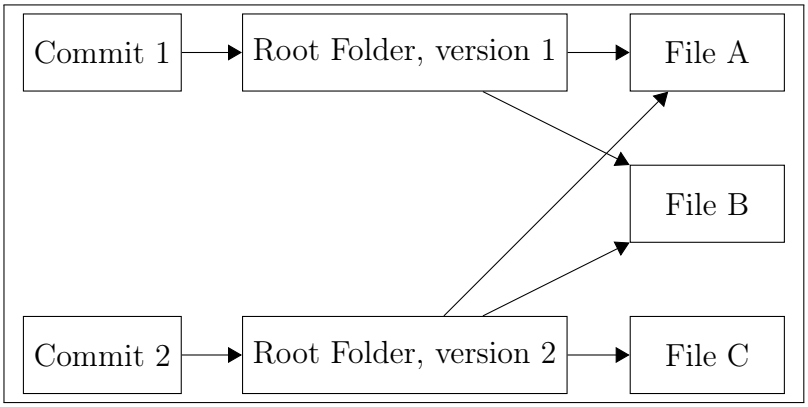


Figure 4.6: Two commits in the Snapshot Reference Storage model. Commit 1 references a folder ("Root Folder") with two files ("File A" and "File B"). In commit 2, a new file ("File C") has been introduced. For the creation of commit 2, files A and B do not get copied. Instead, they are represented by references to the ones introduced in commit 1.

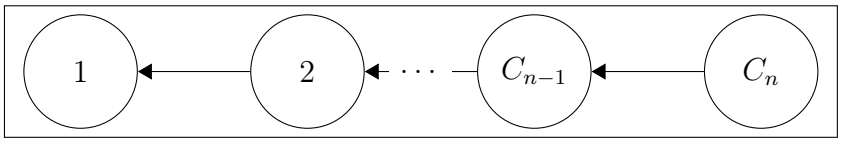


Figure 4.7: An example of the Linear Model used to represent the commit history. The commits, shown as circles, are arranged in a linear way, with arrows indicating the respective parent commit.

working on separate branches.[5]

Now, user B wants to integrate the changes from branch B into branch A. There are multiple ways this could be achieved. One option would be to replace commits C_n and C_{n+1} with C_b and C_{b+1} as shown in figure 4.10. In this case, all changes and the commit history created by user B would now be available on branch A. But commits C_n and C_{n+1} created by user A would be lost, including all changes. Thus, this method is not feasible for any kind of collaboration (or branching in general, even if it is done by only one user), since it does not allow users to combine their work from separate branches, but instead just overrides all changes made on one branch with the changes made on the other, leading to loss of information.
Another option to merge branch B into branch A is shown in figure 4.11: The

[5]Strictly speaking, this violates the definition of the Linear Model, because commit C_{n-1} now has two successors. But in order to allow users to work independently of each other, there needs to be a way to create commits without involving the action of other users. One could say that commit C_{n-1} has one successor per branch, which would again be in accordance with the definition of the Linear Model.

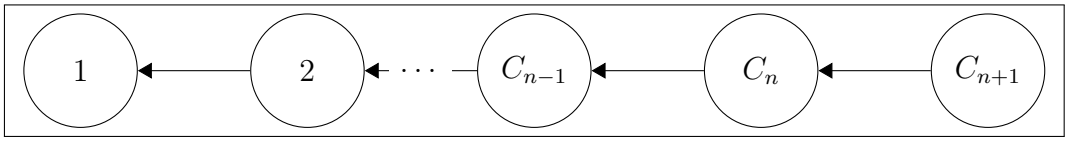


Figure 4.8: The Linear Model after commit C_{n+1} has been added to the commit history shown in figure 4.7.

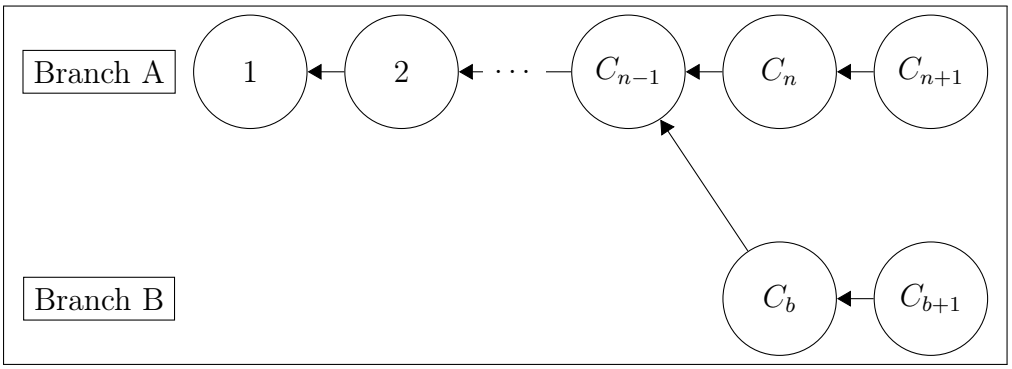


Figure 4.9: The Linear Model with two branches ("Branch A" and "Branch B"). Branch B was created after commit C_{n-1} was added to the commit history. Commits C_n and C_{n+1} have been added to branch A, similar to how it was done in figure 4.8. Commits C_b and C_{b+1} are part of branch B.

commits on branch A are left unchanged. All commits on branch B are combined into one commit C_{b*}. This commit includes all changes that were made on branch B, but it is designed in a way that it also respects the changes made in commits C_n and C_{n+1}. C_{b*} is added to branch A as the latest commit, with C_{n+1} being its parent. The advantage of this approach, compared to the first one, is that changes performed on both branches are not lost. However, the actual commit history on branch B (i.e. commits C_b and C_{b+1}) is not referenced by C_{b*}. Thus, it is not possible to retrace the changes that were made by user B. In this example, this is certainly not a fundamental loss of information. However, if a larger number of commits was made on branch B, probably even hundreds of commits, then it might be useful for users to be able to follow the exact history of changes made on branch B. With this model, the information that commit C_{b*} is based on all the commits in branch B, would simply be lost, and it would appear as though commit C_{b*} was just a single set of changes introduced on branch A.

Figure 4.12 shows a way to make it possible to retrace the historic changes that led to C_{b*}: Instead of only referencing C_{n+1} as its parent, C_{b*} also references C_{b+1} as an additional parent. Thus, C_{b*} has not only one, but two parents. Now, it is possible to follow the changes that were made on both "paths" leading to C_{b*}. This is exactly what the DAG Model allows to define: Instead of a linear history where every commit may only have up to one parent, the DAG Model allows an

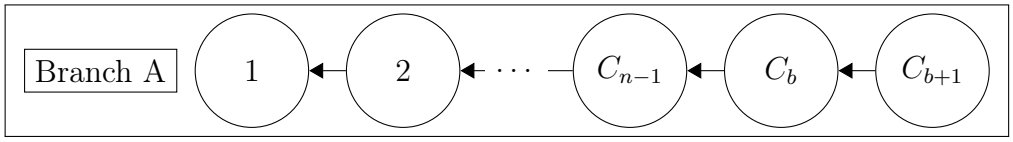


Figure 4.10: A simple idea how to merge the changes (shown in figure 4.9) from branch B into branch A: Commits C_n and C_{n+1} got removed from branch A, and commits C_b and C_{b+1} got added to branch A.

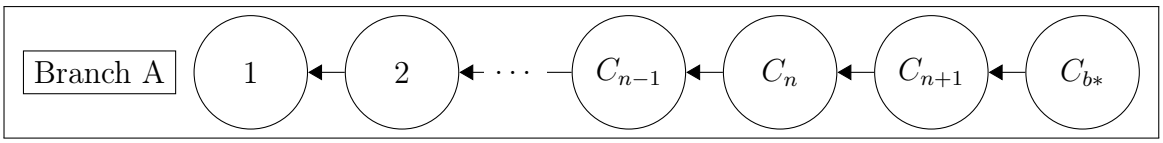


Figure 4.11: An alternative to the approach shown in figure 4.10: Instead of removing commits C_n and C_{n+1}, the two commits have remained unchanged. A new commit C_{b*} was created and contains the changes from branch B in a way that is compatible with the state at commit C_{n+1}. It is not possible to determine that C_{b*} uses commits C_b and C_{b+1} as a basis.

unlimited number of parents for every commit. It should be noted that, as the name implies, a DAG does not allow any cycles. Hence it is not allowed for a commit to be (transitively) referenced by itself in the history, i.e. a commit may not be the ancestor of itself when defining the commit history graph. Technically, it is not necessary that one branch is merged into another - a merge is essentially performed between two commits, so in the example described before, it would also have been possible to merge commit C_b (which is not the latest commit found in branch B) into branch A.

The DAG Model allows to model the commit history in a similar way as the Linear Model: For every commit, it is possible to determine its historic development. But unlike the Linear Model, the DAG Model makes it possible to perform merges while retaining the history of every merged commit, i.e. without any loss of information. This makes the DAG Model perfectly suited for the handling of the commit history with all of the required features the Hub VCS shall provide. Because of this, the DAG Model has been selected as the history model for the Hub VCS.

Since files and folders are stored using the Snapshot Reference Storage model, a commit can directly reference (a root folder of) a snapshot. This also shows one of the strengths of the Snapshot Reference Storage model: If the Delta Storage model was used, then it would not be clear how deltas should be handled. Since a commit in the DAG Model can have multiple parents, a delta would have to be able to cover differences not only between one "source" and one "target" commit, but also between multiple source commits and one target. This would

most likely introduce additional complexity, especially when the content of a tree item for a particular commit shall be fetched. In the Snapshot Reference Storage model, commits simply directly point to a file/folder tree snapshot, no matter how many parents a commit has. To fetch the content of a tree item for a particular commit, no additional (computational) complexity is added, so the complexity for retrieving this tree item remains at ($\mathcal{O}(1)$).

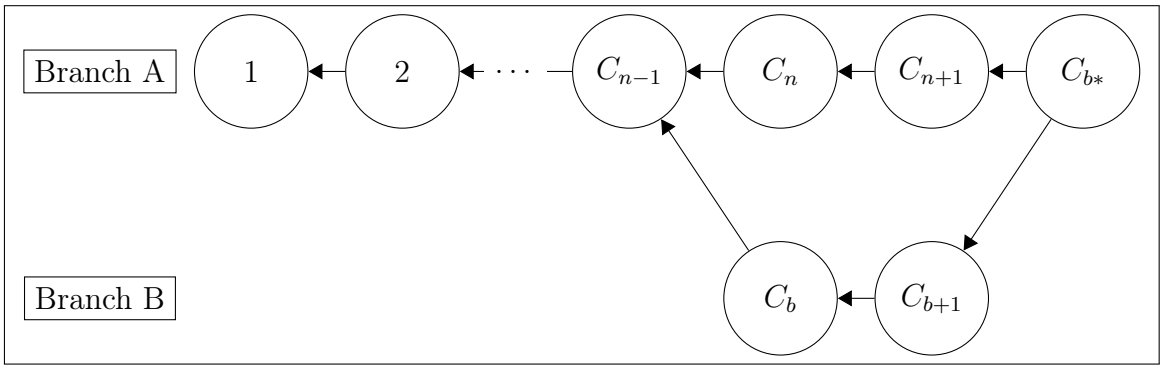


Figure 4.12: An example of the DAG Model: Similarly to figure 4.11, a commit C_{b*} was created. But unlike in the Linear Model, C_{b*} has references to two parent commits (to C_{n+1} and to C_{b+1}). This makes it later possible to retrieve the entire commit histories of C_{n+1} and C_{b+1}, so no information is lost because of the merge.

4.3.4 Updated Logical Model

Figure 4.13 shows an updated overview of the Logical Model. Compared to the original model shown in figure 4.1, two important missing parts have been added:

- The commit history is now modelled using a **N:M** relationship between commits. A commit can have multiple parent commits, and a commit can be the parent of multiple commits.
- The Snapshot Reference Storage model now implements the requirement of storing files, folders, and deltas. Folders are modelled using a **N:M** relationship: One folder can be the parent of multiple other folders, and one folder can also be part of (i.e. referenced by) multiple (parent) folders. The connection between files and folders is also modelled using a **N:M** relationship: One folder can contain multiple files, and one file can be part of (i.e. referenced by) multiple folders.
  Deltas are not directly stored since the model only stores snapshots. However, it is possible to compute the delta between two arbitrary commits by comparing the respective file/folder tree snapshots.

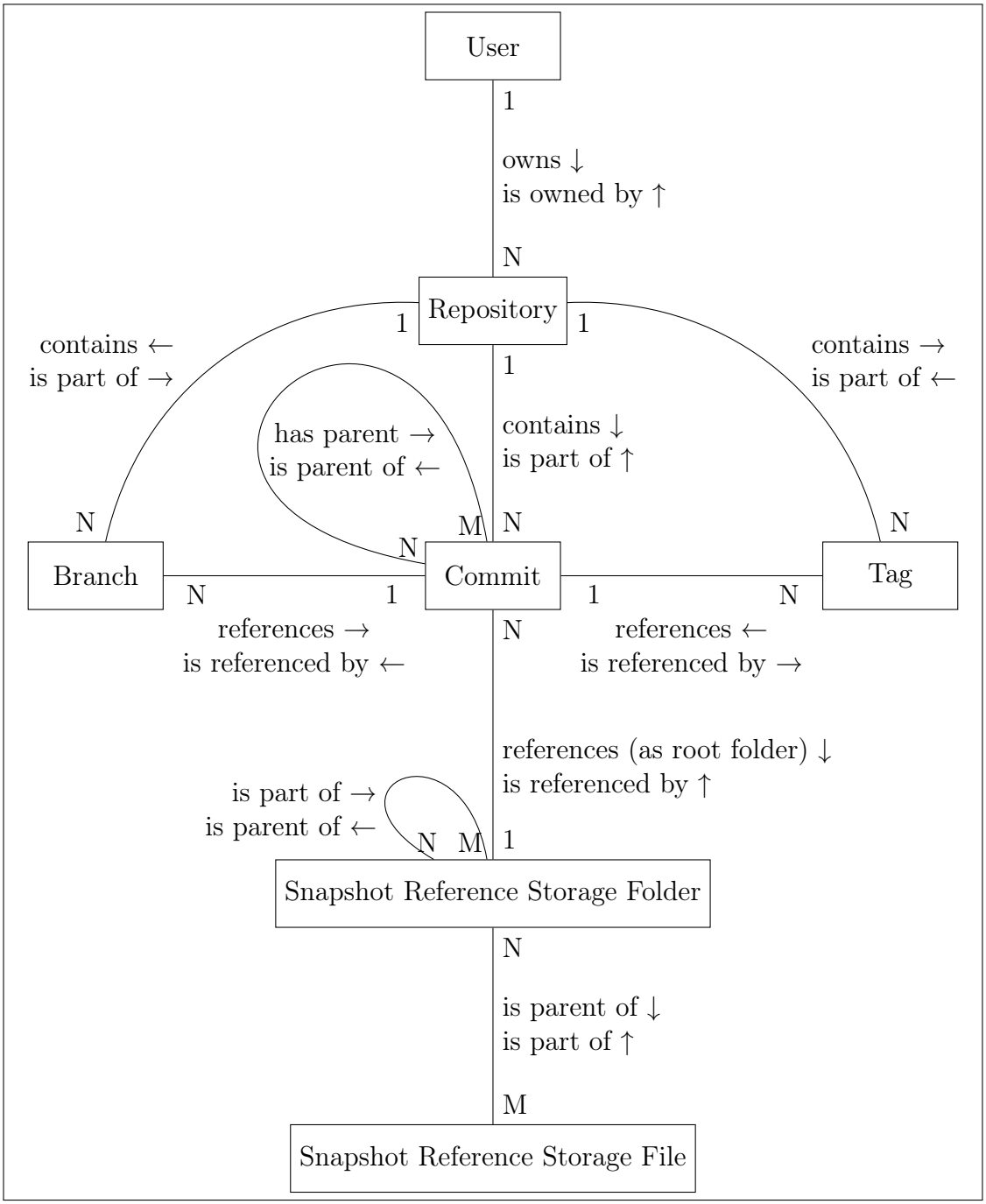


Figure 4.13: An updated overview of the Logical Model, based on the overview shown in figure 4.1. A "Snapshot Reference Storage Folder" is a folder in the Snapshot Reference Storage model, and a "Snapshot Reference Storage File" is a file in the Snapshot Reference Storage model. A commit always references one folder (the root folder) of a file/folder tree in the Snapshot Reference Storage model.

5 Architecture

This chapter provides an overview of the technical structure of the Hub VCS. While chapter 4 provides an implementation-independent overview of the most relevant concepts and decisions needed for the Hub VCS, this chapter outlines the concrete architecture used in the implementation of the Hub VCS.

5.1 Foundation of final design

The decisions regarding the final system design are based on the functional and non-functional requirements as described in chapter 3, as well as the main conceptual decisions as described in chapter 4.

Since the Hub VCS is meant to be structured as a CVCS, the Hub VCS can be divided into a client and a server part. The client(s) are not meant to persist any data - instead, the server is the "single source of truth". That means that operations like committing or checking out a particular commit must include some kind of communication between a client and the server. Technically, it is not necessary that every interaction a user performs on a client application directly makes the client communicate with the server, since it is also possible that certain actions are performed "locally" and only later on synchronized with the server. However, every change that shall be persisted must in the end involve communication with the server. This makes the server the central and most important part of the Hub VCS.

In the following, the server will be referred to as "backend".

5.2 Backend

As described in requirement N-2, the backend is built using TypeScript as the main programming language. This makes it particularly easy to reuse parts of the program in other software applications developed in the context of the

JValue Project. The compiled TypeScript code is executed in Node.js[1], while dependencies to (external) software packages are managed using npm[2].

Since the main interactions of the user are meant to be performed through a web-based frontend as defined in requirement N-1, the backend must be accessible through an HTTP API. To provide this API, the backend uses Express[3], which is a library that, among others, makes it possible to "listen" and react to HTTP requests made by clients.

A very important part of the backend of the Hub VCS is its persistent data storage: Here, all repositories, branches, commits, and more, are stored. The decision was made to use a relational database system like PostgreSQL[4] or MariaDB[5] as the only persistent storage system, i.e. the one and only point where all data is actually stored and managed. Unlike the storage of data using regular files in a file system, relational database systems typically make it possible to quickly access and aggregate data even in scenarios with very complex relationships, allow developers to avoid race conditions using locking mechanisms, and ensure high data consistency with the definition of precise schemas, constraints, and the execution of queries in ACID-compliant transactions[6]. PostgreSQL has been selected as the database system to be used in the Hub VCS, since it is also used in other software applications in the context of the JValue Project.

5.3 Clients

All software applications that communicate with the backend using the HTTP API are referred to as "clients" in this context. What all clients have in common is that they are built using TypeScript, and that npm is used to manage the (external) libraries they depend on. For the Hub VCS, two web applications (User Interfaces (UIs)) that shall later be used by end users, an "Importer" for performance testing, and a suite of system tests have been developed that can all be classified as clients in this setting. In the following, these software applications

[1] https://nodejs.org/en/, accessed on June 21, 2022
[2] https://www.npmjs.com, accessed on June 21, 2022
[3] https://expressjs.com, accessed on June 21, 2022
[4] https://www.postgresql.org, accessed on June 21, 2022
[5] https://mariadb.org, accessed on June 21, 2022
[6] Transactions make it possible for one database client to execute multiple operations in complete isolation from other clients. "The concept of a transaction [...] requires that all of its actions be executed indivisibly: Either all actions are properly reflected in the database or nothing has happened." (Haerder & Reuter, 1983) For this purpose, Haerder and Reuter (1983) have introduced four properties that must be provided by a database system: Atomicity, Consistency, Isolation, and Durability - those properties are also referred to as "ACID" properties.

will be described in more detail.

5.3.1 User-facing clients (UIs)

To make it possible for end users to interact with the Hub VCS, two UIs have been implemented: The "Contributor Frontend" and the "Viewer Frontend". As described in requirement N-2, both applications are built using Vue.js, which is a framework for the creation of web applications.

From a technical point of view, it would certainly also have been possible to integrate all features into one single frontend. However, as described in chapter 2, there are two major projects the Hub VCS will later communicate with (or be integrated into): The DEWB and the JValue Hub. While the DEWB focuses on content editing, the JValue Hub is focused on the presentation and exploration of information. To provide a similar distinction between "writing" and "reading", the decision was made to develop not one, but two frontend applications.
The Contributor Frontend is meant to demonstrate the "writing" features of the Hub VCS: It allows to create repositories, branches, tags, commits (including files and folders), and more. Similar to how the DEWB is meant to edit content, the Contributor Frontend makes it possible to make changes to a repository. Thus, in the same way as the DEWB allows to edit data source configurations, the Contributor Frontend makes it possible to create and update information stored in the Hub VCS.
The Viewer Frontend on the other hand makes it possible to explore repositories: It provides features to view the history of a file or folder, to compare two revisions (i.e. to visualize the diff between two commits), and more. Hence, the Viewer Frontend can be seen as an analogy to the JValue Hub.

The Contributor Frontend and the Viewer Frontend are both applications that provide a non-trivial set of features: Among others, the Contributor Frontend needs to provide features to edit files, to make changes to folder trees, to create commits (ideally with the ability for the user to see which changes he or she is about to commit), to perform updates to the working copy, and even to create merge commits. The Viewer Frontend on the other hand needs to provide features for the exploration of commit histories, a suitable visualization method for diffs, and more. To be able to properly structure and design these two applications,

Mockups and Prototypes[7] have first been created in Figma[8]. Figma is a tool that, among others, is used in the field of UI and User Experience (UX) Design to create the visual parts of an application in the form of Mockups, without having to actually write software code. This allows the designers to focus their work on the general layout and feature composition, without getting distracted by the complexity of actual program code. Compared to directly implementing a particular part of a frontend in software code, the creation of Mockups leads to much faster iterations. In Figma, it is also possible to add basic interactivity to these Mockups (e.g. by simulating what happens when a particular button is pressed), which turns these Mockups into Prototypes.
Figure 5.1 shows a Mockup created in Figma that presents how a view for editing files and folders could look like in the Contributor Frontend. Even though this looks like a screenshot from an actual software application, it is merely a composition of visual primitives like rectangles and text. In the Figma Prototype, it is possible to click on selected regions in order to simulate how a user might later navigate through the real application. Figure 5.2 shows another Mockup of the Contributor Frontend: Here, a commit view is shown, i.e. a page that allows the user to see all the changes he or she has made to the working copy, and to select which files shall be part of the next commit.

After the content that had been identified as relevant for the Contributor Frontend and Viewer Frontend had been built in Figma, the actual Vue.js frontend applications were implemented with the Figma layouts as a basis. Since the Prototypes already provide a very realistic preview of the interactions a user shall be able to perform, the structure of the components implemented in Vue.js could be derived from the "blueprints" designed in Figma.

5.3.2 System test suite and Importer

Two other pieces of software communicate with the backend: A suite of system tests, and the Importer. Similar to the user-facing clients, the system tests and Importer only communicate with the backend via the HTTP API. Both software packages are used to test the backend.

The system test suite is based on Jest[9], which is a testing framework for JavaS-

[7]Some sources like Rivero et al. (2014) treat the words "Mockup" and "Prototype" as synonyms. In this thesis however, we say that a Mockup is static (e.g. an image) - it does not react to any type of user interaction. Once some kind of interactivity is added to a Mockup (e.g. navigating to a different Mockup when clicking on a certain region in the first one), the Mockup turns into a Prototype. Thus, a Prototype is an interactive Mockup. The scope of interactivity is irrelevant - even if only one small button or the like is simulated, the Mockup turns into a Prototype.

[8]https://www.figma.com, accessed on June 21, 2022

[9]https://jestjs.io, accessed on June 21, 2022

Figure 5.1: A Mockup created using Figma, showing how a view for editing files and folders could look like in the Contributor Frontend.

cript applications. Every time the system test suite is executed, a backend with a "fresh" (i.e. empty) PostgreSQL database gets started. Then, the actual tests are run. All tests work in a very similar way: They perform requests to the HTTP API and check if the response from the server matches the expected response. For instance, a test for the commits API first sends a request to create a commit, and then sends another request to retrieve the commit that has just been created. The data returned by the second request are matched against the data sent in the first request. If there is a mismatch, the test fails. Such a pattern is used among all tests and ensures that the HTTP API fulfills its specification, and that edge cases are properly handled.

The system test suite is meant for automatic testing: After making a change to the code base of the backend, the system tests should be executed and once the execution has been completed, the developer only needs to check if the tests all pass or if an error is indicated. For the development of the Hub VCS, this has been automated using a Continuous Integration Pipeline, which executes the tests whenever a new commit gets pushed to the repository the development takes place in. The Importer, on the other hand, is used for manual testing, and is mostly meant for performance evaluation: The Importer is a Command Line Interface (CLI) application that makes it possible to import parts of a Git repository into the Hub VCS by performing requests to the HTTP API. This is especially useful to determine whether the backend of the Hub VCS is capable of

Figure 5.2: A Mockup created using Figma, showing how a view for the creation of commits could look like in the Contributor Frontend.

handling a large number of commits, files, and folders.

A user of the Importer first needs to select a Git repository he or she wants to import into the Hub VCS. There are no restrictions regarding the structure of this repository - any Git repository can be handled by the Importer. Such a Git repository can of course be created manually by the user. However, in order to test whether the system is able to handle hundreds or thousands of commits, and/or a large number of files and folders, the user typically clones an already existing (public) repository from a platform like GitHub or GitLab.
After the Git repository has been selected in the Importer CLI prompt, the user has to provide a short list of additional information, such as the "target" repository on the Hub VCS the import should be performed into.
Then, the Importer determines which commits shall be imported from the commit history: It loads the commit history of a branch that has been chosen by the user, and selects one linear "path" through this history. That means: If a commit in the Git repository has more than one parent, then only one of these parents will be selected by the Importer to be sent to the Hub VCS.[10]
Finally, for every commit in the linear history, starting at the oldest one, the

[10]This technically means that parts of the commit history in the Git repository are ignored. However, since the Importer is not meant to provide any kind of interoperability between Git and the Hub VCS, but is rather only meant for testing purposes, this simplified history import does not have any major negative impact on the testing process. On the contrary, it makes the imported history easier to predict, and greatly simplifies the overall import procedure.

Importer creates a commit in the Hub VCS using the HTTP API. Optionally, after every imported commit, the Importer can perform "Reverse Tree Validation", which means that the file/folder tree for the just created commit is again fetched from the backend and compared to the expected structure.

Once all commits have been imported, the user can perform the manual tests he or she intended to perform. For instance, the user can load the commit history using the Viewer Frontend and observe whether the response time of the backend is acceptable, and whether the returned data matches the expectations.

This kind of testing should be seen as complementary to the system tests: While the system tests provide a strict set of tests and expected results (which can also automatically be executed), the manual tests allow to answer questions related to approximate response times, and whether the Hub VCS is able to handle real-world repository sizes. Since the Importer allows to create large numbers of commits, files, and folders, it can also be utilized for development purposes, especially as an (informal) way to test potential performance optimizations.

5.4 Final system structure

As described before, all of the clients, as well as the backend, are built using TypeScript, while their dependencies are managed using npm; each of these software applications is a separate npm package. In order to allow important parts of code to be reused, an additional Shared package has been introduced. Among others, the Shared package contains models and helper functions for dealing with data structures used in the Hub VCS, as well as information about the endpoints provided by the backend. To properly orchestrate all npm packages, and especially to make it possible to easily reference the Shared package in other parts of the system, the decision was made to use npm workspaces[11], which is a feature provided by npm to orchestrate multiple packages and their dependencies.

Figure 5.3 shows the structure of the packages the Hub VCS consists of: There are four client packages (Contributor Frontend, Viewer Frontend, System Tests, and Importer), one server-side package (the Backend package), and the Shared package. The client packages are independent of each other - they only communicate with the backend, but never directly with each other. All client packages and the Backend package use ("import") models, helper functions, and the like, from the Shared package. A side effect of this system structure is that it makes it almost trivial to implement new clients, since most of the complex business logic is already encapsulated in the Shared package and in the backend.

[11] https://docs.npmjs.com/cli/v8/using-npm/workspaces, accessed on June 21, 2022

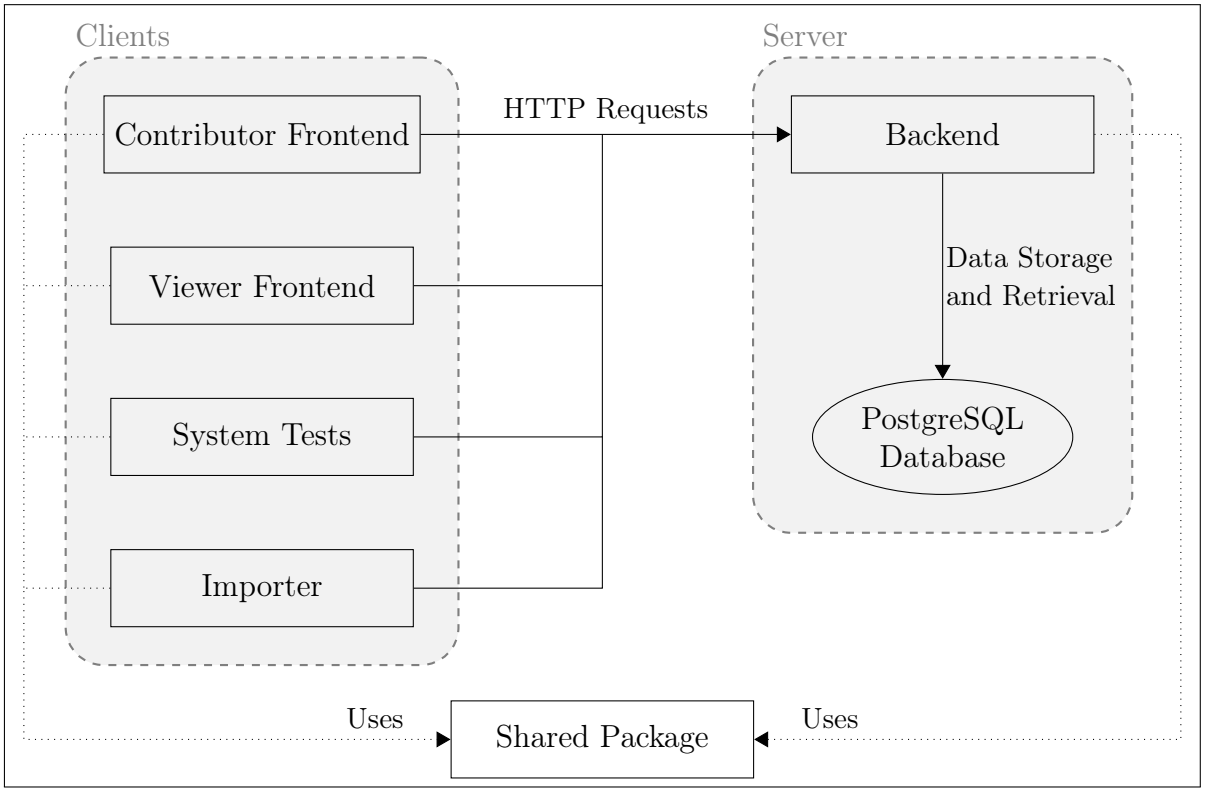


Figure 5.3: The final system structure: There are four client packages, one server-side package (the Backend package), and the Shared package.

6 Implementation

This chapter highlights details about interesting and particularly important parts of the final system developed for this thesis. These implementation details are a result of, and based on, the theoretical and practical foundations described in chapters 4 and 5. Since the most relevant logic of the VCS can be found in the backend, this chapter focuses mostly on the server-side part of the application.

6.1 Database schema

The most relevant implementation details are centered around, or highly influenced by, the database schema. Thus, an overview of the database schema is presented hereinafter.

6.1.1 Storage of files and folders

Files and folders are stored using hash trees, following a similar mechanism as used in Git[1]: To store a new file, the hash of the content is computed. If a file with the same hash already exists, then nothing else needs to be done (it can simply be referenced by its hash). If a file with the same hash does not exist, then the file is stored, using the hash as an identifier. Folders are handled in the same way, with the difference that the hash is based on the hashes of their children. The major difference between the hash trees used in Git and the Hub VCS is their scope: In Git, repositories exist independently of each other. Thus, hash trees cannot be "shared" between multiple repositories. If, for example, a file F is stored in repositories A and B, then a copy of F exists twice (once in repository A, and once in repository B). In the Hub VCS however, hash trees are reused across repositories. If file F is stored in repositories A and B, then this file is actually only stored once in the storage system of the Hub VCS (and

[1]The storage mechanisms used by Git are roughly described by Chacon and Straub (2022).

referenced twice in this example). This storage system is especially useful if the same file content (or even entire folder) can be found in many repositories.

Figure 6.1 shows how files and folders ("tree items") are stored in the PostgreSQL database. Files are stored in table ***tree_files***. This table contains three columns: ***hash***, ***mode***, and ***content***. Column ***hash*** contains the hash of the data stored in the other two columns. This hash is used as an identifier; In many parts of the implementation, files are referenced by their hash. ***mode*** can either be *text* or *binary*, while ***content*** is always a text string. To store a binary file, the file must be transformed to a string by the client, e.g. using a Base64[2] encoding. The backend does not define any standard for the encoding of a binary file - the format must be selected by the client(s), depending on the use case. It should be noted that file names or other meta information are not stored in table ***tree_files***.
Table ***tree_folders*** contains information about the **content** of a folder. This table is essentially a mapping from a particular folder hash to its children (i.e. to files and other folders that are stored in the first folder). The table has four columns: ***folder_hash***, ***child_name***, ***child_type***, and ***child_hash***. Column ***folder_hash*** contains the hash of the entire folder, i.e. of all files and folders that are directly referenced by this folder. For every child of a particular folder, a separate row exists in this table. If a folder F has n children, then n rows exist in table ***tree_folders*** to store folder F, all of these rows having the same value in column ***folder_hash***. Column ***child_type*** describes whether a particular folder child is a file or a folder. ***child_name*** contains the file or folder name of the child, and ***child_hash*** contains the hash used to reference the particular child file or folder.

In the following, an example is constructed in order to better illustrate the functionality of these two tables.

Two text files *FileA.txt* and *FileB.txt* shall be stored. These files have the content "This is file A" and "This is file B". Thus, two entries are created in table ***tree_files*** as shown in figure 6.2: For every file, the hash is computed and stored alongside the file mode and content. The hashes shown in this example are not authentic, but are only meant to illustrate the functionality of the system.

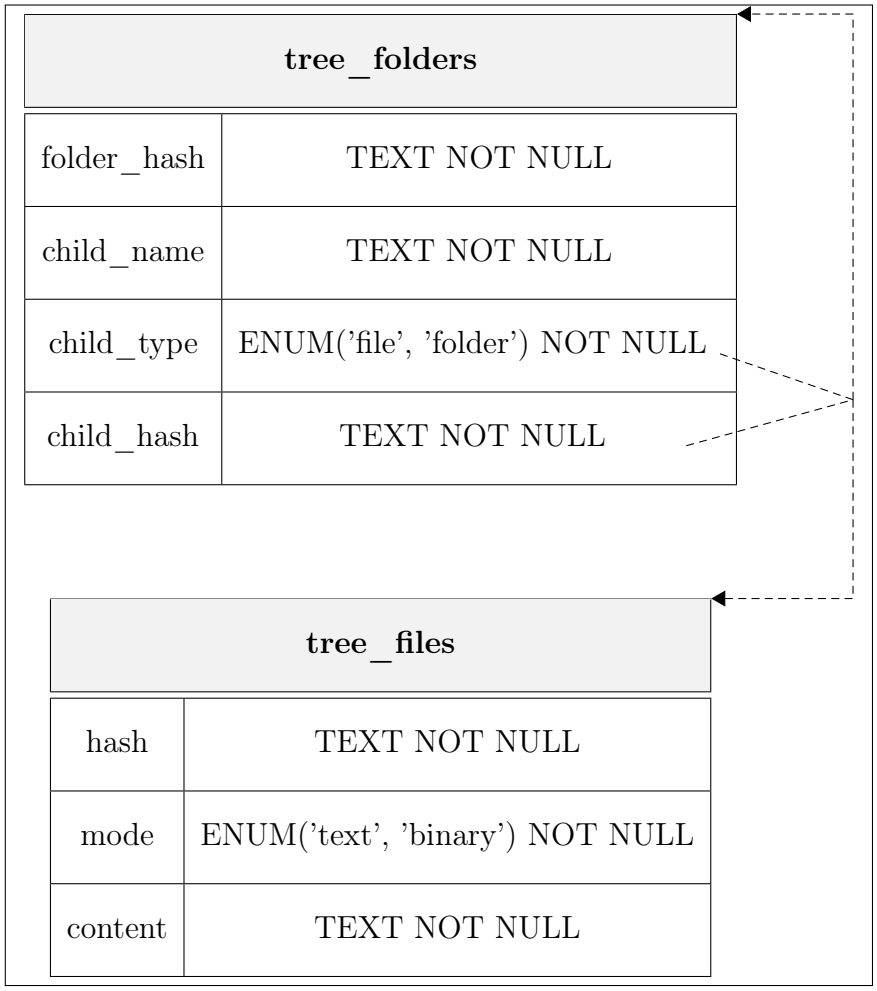


Figure 6.1: The schema in the PostgreSQL database of tables ***tree_folders*** and ***tree_files***. ***tree_folders*** references children that are either folders (thus stored in ***tree_folders*** as well), or files (stored in ***tree_files***).

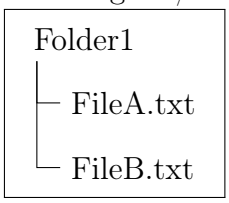
tree_files		
hash	mode	content
8eafa3cace	text	This is file A
97fa7db13f	text	This is file B

Figure 6.2: The content of the two sample files *FileA.txt* and *FileB.txt* stored in table ***tree_files***.

tree_folders			
folder_hash	child_name	child_type	child_hash
f3f722036d	FileA.txt	file	8eafa3cace
f3f722036d	FileB.txt	file	97fa7db13f

Figure 6.3: The content of folder *Folder1* stored in table ***tree_folders***.

Now these two files shall be stored in a folder called *Folder1*, leading to the following file/folder structure:



In order to store the content of *Folder1*, two entries need to be created in table ***tree_folders***, defining files *FileA.txt* and *FileB.txt* as the children of *Folder1*. This is illustrated in figure 6.3. One aspect of the storage structure now becomes apparent: The name of a tree item (i.e. of a file or folder) is always stored "one level up" - the names of the two files of this example have only been stored once they have been embedded as children of *Folder1*. Since *Folder1* currently has no parent folder, its name is not stored. This structure makes it possible to rename a file without having to store a second copy of it: If *FileA.txt* was now renamed to *AnotherFileA.txt*, the entries in table ***tree_files*** would not have to be altered.

[2]https://developer.mozilla.org/en-US/docs/Glossary/Base64, accessed on June 21, 2022

tree_folders			
folder_hash	child_name	child_type	child_hash
f3f722036d	FileA.txt	file	8eafa3cace
f3f722036d	FileB.txt	file	97fa7db13f
a40d4d3725	Folder1	folder	f3f722036d
a40d4d3725	AnotherFileA.txt	file	8eafa3cace

Figure 6.4: The content of folder *RootFolder* stored in table ***tree_folders***. The content of folder *Folder1* as shown in figure 6.3 remains unchanged.

To conclude this example, another folder called *RootFolder* is introduced. This folder has two children: *Folder1*, and a new file *AnotherFileA.txt*, which has the exact same content as *FileA.txt*. The file system is now structured as follows:

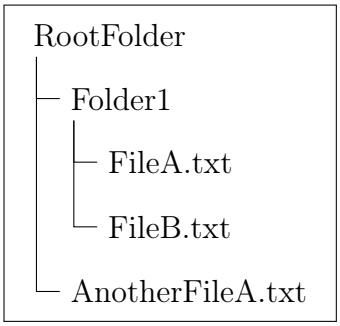


Figure 6.4 shows the updated content of table ***tree_folders***: Two entries (the children of *Folder1*) have remained unchanged. Two entries (the two children of *RootFolder*) have been added - one entry holds *Folder1* by defining ***child_type*** as "folder" and using the hash of *Folder1*, the other entry holds *AnotherFileA.txt* by setting ***child_type*** as "file" and using the same hash that has previously been used for *FileA.txt*. Table ***tree_files*** remains unchanged, since the only added file (*AnotherFileA.txt*) has the same content and thus the same hash as an already existing file (*FileA.txt*).

6.1.2 Storage of commits, repositories, and more

Figure 6.5 shows how commits are stored: Table ***commits*** contains the actual commits, including meta information like a title, description, and more. Commits are identified using a unique number (*ID*). Column *tree* is used to reference an entry (root folder) in table ***tree_folders***, using the hash of the entry. Table

commit_parents contains information about commit parents, i.e. a mapping from a commit to its parents. Thus, table ***commit_parents*** contains the actual DAGs of commits.

Figure 6.6 illustrates how all of the aforementioned tables reference each other. Additionally, it shows three additional tables: ***repos***, ***branches***, and ***tags***. ***repos*** contains information about the repositories stored in the Hub VCS. Just like with commits, repositories are referenced using a unique number (*ID*). Every repository contains multiple branches and tags. Branches and tags are also referenced using an *ID*.

One aspect is only shown indirectly: Similar to the hash trees, commits do not necessarily belong to only one repository. Instead, a particular commit can be part of multiple repositories. Because of this, the ***commits*** table does not contain a column that references a repository. Whether a commit C belongs to a certain repository R can only be determined using the following technique:

1. Load all branches and tags of repository R. In the following, the set of branches and tags part of R is called BTR.
2. Fetch the commits that are referenced by BTR. Hereinafter, the set of these commits is called BTC.
3. Is one of the commits in set BTC equal to commit C? Then we know that C is part of repository R. Nothing needs to be done anymore.
4. None of the commits in set BTC was equal to commit C? Then recursively extract all commit ancestors (i.e. the commit parents, the parents of the parents, and so on) of the commits in BTC. Once C is found in the recursive extraction of commit ancestors, we know that C is part of R and the computation can be stopped. If, however, the recursive check did not find commit C, then C does not belong to R.

This model allows to implement complex branching techniques across multiple repositories without having to copy commits or the like. We say that the Hub VCS acts as a "Super Repository": Technically, repositories are merely a layer on top of branches and tags, instead of acting as hard boundaries that strictly separate commits belonging to one repository or another. Such a feature was not described in the requirements defined in chapter 3, but has been identified as a feature that might be needed in the Hub VCS in the future.

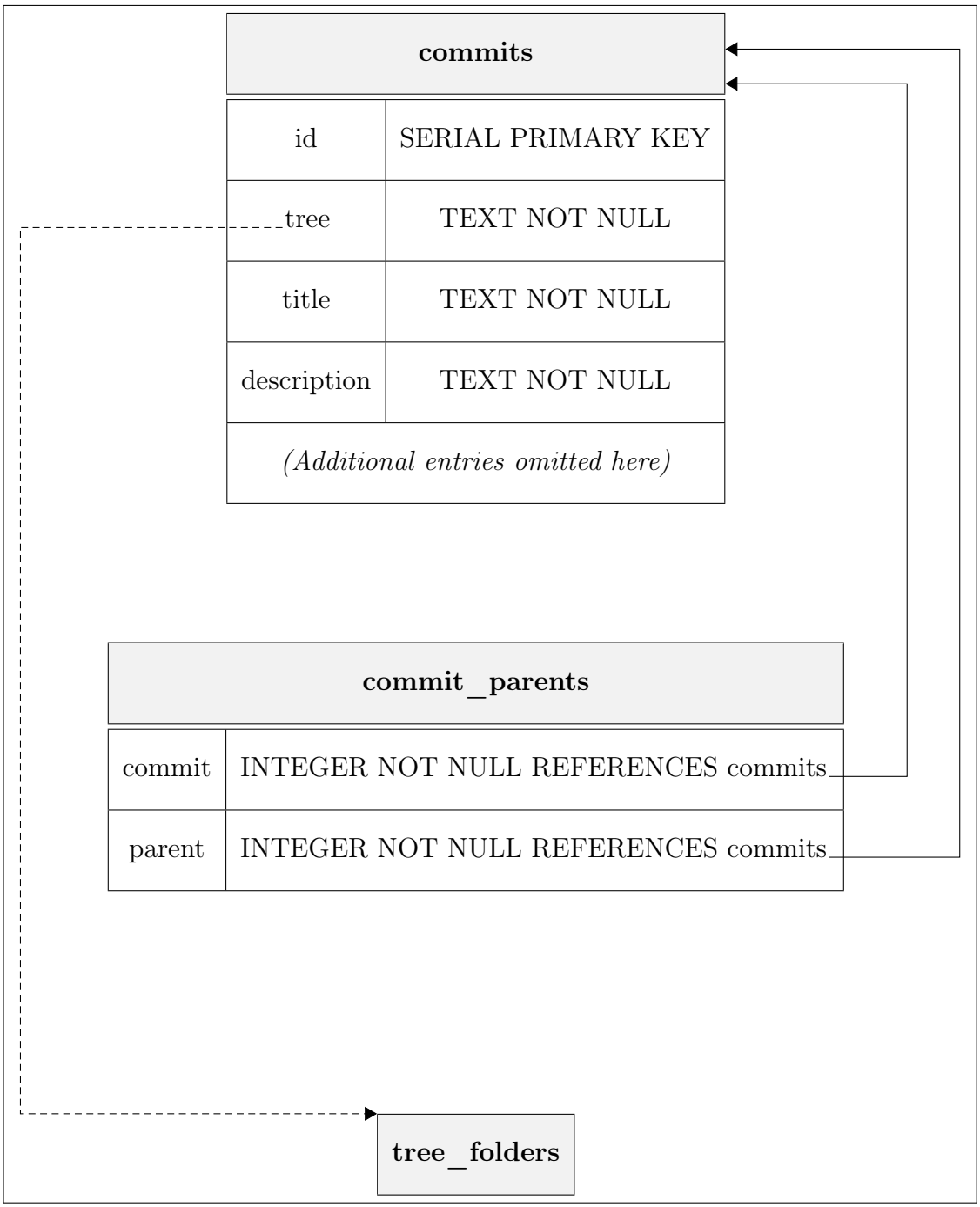


Figure 6.5: The schema in the PostgreSQL database of tables ***commits*** and ***commit_parents***. ***commits*** contains the actual commits, while ***commit_parents*** holds information about the commit parents, thus creating the commit DAGs.

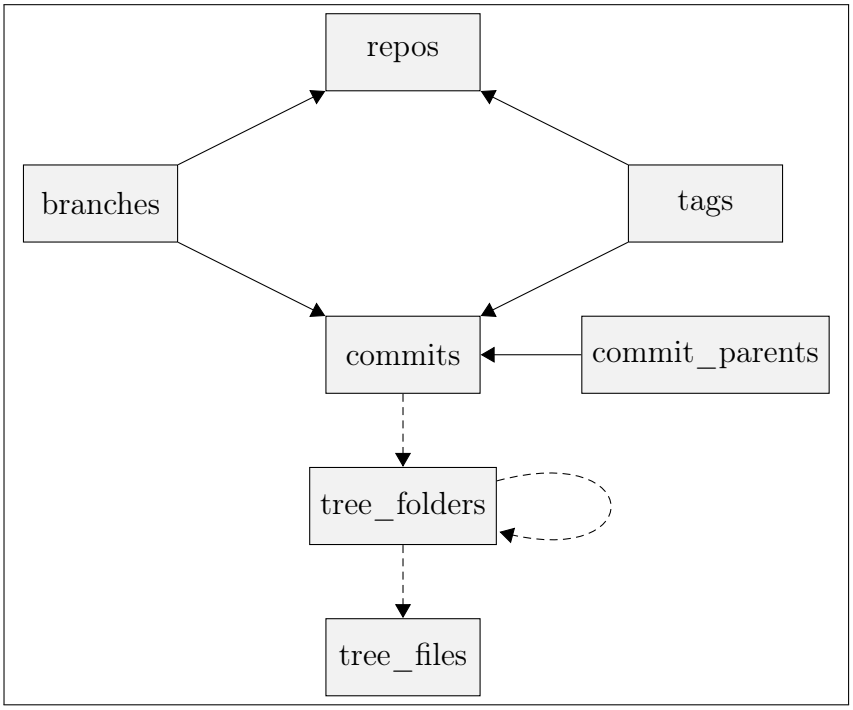


Figure 6.6: An overview of the database tables used to store information about repositories, commits, and the like. The edges drawn in this diagram illustrate the references between individual tables. For instance, table ***branches*** has a column *repo* to indicate which repository a branch belongs to. Thus, the edge is drawn from ***branches*** to ***repos***. A dotted edge is drawn if the reference is not maintained using the PostgreSQL *REFERENCES* keyword.

6.2 Creation of commits

In the following, the most important steps in the creation of a regular commit are explained.
A client can create a new commit by sending an HTTP *POST* request to the backend, using the following endpoint:
`/repos/{repoId}/commits`
Here, `{repoId}` is the *ID* of the repository, which is a simple number.

The request body contains the following information:

- `title`: The title of the commit, which can for example contain a short description of the changes the commit introduces.
- `description`: The description of the commit, which can for instance be used to describe the changes of the commit in more detail.
- `branchId`: The *ID* of the branch the commit shall be created in. It is not possible to create a commit outside a branch.
- `parentCommitId`: The *ID* of the commit the new commit is based on. It is only possible to create a commit if `parentCommitId` matches the head of the given branch. If `parentCommitId` is not up-to-date (i.e. it does not match the head of the given branch), the request will be rejected. This is done to avoid race conditions with other clients.
- `diff`: The changes (added, updated, and deleted files) compared to the state at `parentCommitId`.

Figure 6.7 shows an example of a request body used to create a new commit. The backend expects the request body to be encoded as JSON. In this example, a commit on branch 5 is made, based on commit 7. Three changes are applied to the file/folder tree:

- A new file */file2.txt* is added.
- The content of */file1.txt* is modified.
- File */documents/important-content/content.txt* is deleted.

All files in the diff are referenced by their path. In this example, the deleted file was called *content.txt* and was stored in folder *important-content*, which in turn was stored in folder *documents*.

```
{
  "title": "Add important changes",
  "description": "This commit introduces an important update which resolves multiple issues discussed with the team.",
  "branchId": 5,
  "parentCommitId": 7,
  "diff": {
    "addedFiles": [
      {
        "path": "/file2.txt",
        "file": {
          "fileMode": "text",
          "content": "This is the content of file2.txt"
        }
      }
    ],
    "updatedFiles": [
      {
        "path": "/file1.txt",
        "file": {
          "fileMode": "text",
          "content": "This is the content of file1.txt, which has been updated"
        }
      }
    ],
    "deletedFiles": ["/documents/important-content/content.txt"]
  }
}
```

Figure 6.7: An example of a request body used to create a new commit.

In the backend, the request is handled as follows:

1. Start a new database transaction. Hereinafter, if any operation fails, a `ROLLBACK` command is executed to discard all potential changes.
2. Check if the repository (as provided in the URL) exists and if the user has proper access permissions to make changes in the repository.
3. Check if the requested branch exists and whether it is part of the repository.
4. Acquire an exclusive lock on the branch to avoid race conditions.
5. Check if the requested parent commit ID is matching the head of the branch.
6. Load the Base Hash Tree. This is the hash tree that is referenced by the parent commit.
7. Calculate the hashes of all added and updated files that can be found in the diff provided in the request payload. Based on these hashes, build the "Hashed Commit Creation Diff", which is a data structure similar to the diff provided in the request payload, but with an additional field called "hash" for all added and updated files.
8. Apply the Hashed Commit Creation Diff to the Base Hash Tree, i.e. add, update, or delete files as requested by the client. The result is the hash tree of the new commit.
9. Write the new files into the database. For this purpose, extract all added and updated files from the Hashed Commit Creation Diff, and execute a database query that inserts all of these files into the database if they did not already exist.
10. Write the new folders into the database. This requires to recursively convert the updated hash tree into a mapping from folder hash to its children. If a particular folder entry already exists in the database, it will simply be ignored.
11. Write the actual commit into the database, referencing the root folder that has been inserted in the previous step.
12. Add an entry to table `commit_parents` to capture that `parentCommitId` is a parent (in this situation: the only parent) of the newly created commit.
13. Update the head of the aforementioned branch to now point to the newly created commit.
14. Run a `COMMIT` statement to end the transaction.
15. Return the *ID* of the new commit to the client.

For writing the files and folders of a commit into the database, only two database queries are required (one for the files, and one for the folders). This is especially important when dealing with large diffs: If the number of queries was dependent on the size of the diff, larger diffs could negatively impact the performance of the commits endpoint, or even of the entire backend. Figure 6.8 shows how files are inserted into the database: First, all added and updated files are extracted from the diff. Then, a list of "rows" is constructed. Finally, a database query is executed that inserts all entries found in the aforementioned list into table ***tree_files***. Since the database schema requires column ***hash*** to only contain unique values, duplicate files will be ignored.

```
interface TableLayout {
  hash: string;
  mode: 'text' | 'binary';
  content: string;
}

const filesToInsert: TableLayout[] = [];

for (const addedOrUpdatedFile of
hashedCommitCreationDiff.addedOrUpdatedFiles) {
  filesToInsert.push({
    hash: addedOrUpdatedFile.hashedFile.hash,
    mode: addedOrUpdatedFile.originalFile.fileMode,
    content: addedOrUpdatedFile.originalFile.content,
  });
}

const sqlQuery = `
  INSERT INTO tree_files(hash, mode, content)
  SELECT * FROM json_to_recordset($1::json) AS (hash TEXT, mode
tree_files__mode, content TEXT)
  ON CONFLICT DO NOTHING
`;

const insertFilesQueryResult = await dbClient.query(sqlQuery, [
  JSON.stringify(filesToInsert),
]);
// Handle errors based on the query result.
```

Figure 6.8: A simplified version of the TypeScript code used to persist the files that are part of a commit.

6.3 Retrieval of diffs

A client can request the diff between commits by performing an HTTP *GET* request to the `/diff` endpoint of the backend. This endpoint expects two query parameters: `base` and `target`. These are used to identify the two commits that shall be compared. Both parameters are "Fully Qualified Commit References", which is a format designed for the API of the Hub VCS to identify commits, branches, or tags, together with their repositories. For instance, to compare the head commit of branch 2 in repository 1 with the commit referenced by tag 7 of repository 3, a request using the following URL can be used:
`/diff?base=repo-1-branch-2&target=repo-3-tag-7`
Hereinafter, the explanation will be limited to the comparison of two directly specified commits, i.e. of requests that directly reference commits in the `base` and `target` query parameters. For example, a request to compare commit 5 of repository 1 with commit 8 of repository 3 needs to be performed using the following URL:
`/diff?base=repo-1-commit-5&target=repo-3-commit-8`

6.3.1 Basic request processing

The backend handles a request to the `/diff` endpoint as follows: First, a check is made whether the user is allowed to access the *base* and *target* repositories. Then, it is necessary to determine whether the two commit *IDs* actually belong to the respective repositories. This is done using a recursive database query[3]. Figure 6.9 shows a recursive query that checks whether the commit ID specified in parameter *$1* is part of the repository specified in parameter *$2*. To do so, the query traverses all ancestors of commit *$1* in the commit graph. The query returns a single row with a boolean value, indicating whether commit *$1* is part of repository *$2*.

[3]See https://www.postgresql.org/docs/14/queries-with.html#QUERIES-WITH-RECURSIVE, accessed on June 21, 2022

```
WITH RECURSIVE

/* Fetch commit IDs from the branches and tags tables. */
start_commit_ids AS (
  SELECT head AS commit_id
  FROM branches
  WHERE repo = $2

  UNION

  SELECT commit AS commit_id
  FROM tags
  WHERE repo = $2
),

/* The recursive query: Follow the commit parents until we find
what we are looking for. */
rec AS (
  /* Non-recursive part */

  SELECT commit_id
  FROM start_commit_ids

  /* Recursive part */
  UNION

  SELECT parent AS commit_id
  FROM commit_parents
  INNER JOIN rec ON rec.commit_id = commit_parents.commit
  /* This allows us to terminate the recursive query early if we
have found a result. */
  WHERE $1 NOT IN (rec.commit_id)
)

SELECT EXISTS (SELECT * FROM rec WHERE rec.commit_id = $1) AS
commit_exists
```

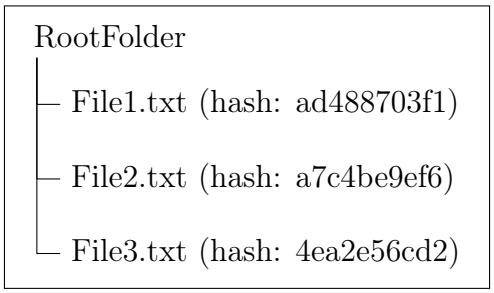
Figure 6.9: A parameterized database query used to determine if the commit specified in parameter *$1* is part of repository *$2*.

After these checks have been performed, the hash trees for the *base* and *target* commits can be loaded. This is also done using a recursive query, followed by a conversion from database rows to an actual tree-like data structure.
Now, it is possible to compare these two trees in order to extract a list of added, updated, and deleted files. The following rules apply:

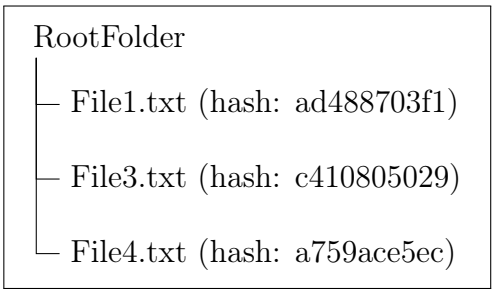
- A file is classified as "added" if it exists in the *target* hash tree, but not in the *base* hash tree.
- A file is classified as "deleted" if it exists in the *base* hash tree, but not in the *target* hash tree.
- A file is classified as "updated" if it exists both in the *base* and *target* hash trees, but has a different hash (and thus a different content and/or file mode).

The comparisons are made based on file paths. As an example, a comparison of the following two hash trees shall be made.

Base hash tree:



Target hash tree:



In this example, *File1.txt* is not part of the list of added, updated, or deleted files, since it exists both in the *base* and *target* hash trees, and its hash is the same in both trees. *File2.txt* is classified as "deleted", since it exists in the *base*, but not in the *target* hash tree. *File3.txt* is an updated file: It exists in both hash trees, but its hash is different when comparing the file in the *base* and *target* hash trees. Finally, *File4.txt* is classified as an added file, since it exists in the *target*, but not in the *base* hash tree.

After the added, updated, and deleted files have been extracted, the information currently available would not be sufficient for a client: At the moment, only the hashes of these files are known, but not their content. Thus, it is then necessary to load the actual file contents from the database. Technically, this step could also have been performed earlier (while or directly after fetching the hash trees from the database). However, this would lead to an unnecessarily high memory consumption, since the *base* and *target* trees would then also include the contents of unchanged files.

6.3.2 Detection of moved files

Now that the added, updated, and deleted files have been extracted and their content is also known, one last step is performed to improve the diff: The detection of moved files, i.e. of files that have been renamed and/or moved into another folder[4]. One option to detect moved files would be to explicitly store information in the database about file renames and the like. However, this would lead to issues when trying to compare two commits that either have no common ancestor (and are thus part of different commit graphs), or have a complex commit graph structure "between them". If for instance a commit has been renamed in one branch, and deleted and later re-introduced in another branch, and these two branches got merged, it would be hard to properly classify this file, as it has both been renamed and deleted. To avoid such ambiguities, the Hub VCS does not explicitly track moves. Instead, heuristics are used that allow to find moves between two arbitrary trees.

The heuristics distinguish two types of moved files: "moved without changes" and "moved with changes". At first, moves without changes are detected. This is done by comparing the added with the deleted files: For every file in the list of added files, the list of deleted files is searched for a matching counterpart. A file must have the same content and the same file mode in order to be considered a match. If a matching counterpart is found, then this is defined as a move without changes, and the pair of files is not considered in any further comparisons. If multiple deleted files match an added file (or vice versa), then an arbitrary selection is made. The only guarantee the heuristic provides in this case is that the selection is "stable": When requesting the exact same diff multiple times, then the pairs of files classified as "moved" are always the same.

It should be noted that the detection of moves operates on lists of files, not on trees. The file path (including the file name) is merely used as an identifier of the respective file. Thus, there is only one list of added files and one list of deleted files that is handled when detecting moves. The "position" of a file in the file/folder tree is not taken into account.

[4]In this part of the implementation, no distinction is made between renamed files and files that have been moved into a different folder. Thus, we always say "moved file".

In the following, an example is constructed in order to better illustrate how moves without changes are detected. The following table shows four added and two deleted files, including their content. To keep the example simple, all files are assumed to be text files.

File name	File content
Added files	
FileA.txt	This is a file
FileB.txt	This is an important file
FileC.txt	This is an important file
FileD.txt	This is a very important file
Deleted files	
FileE.txt	This is an important file
FileF.txt	This is a very important file

Assuming that files are always checked in the order they appear in the table, the heuristic used to detect moves without changes handles these files in the following way:

1. First, *FileA.txt* is handled: The heuristic checks all deleted files for a matching counterpart. However, there is no file with content "This is a file" in the list of deleted files. Thus, *FileA.txt* is not classified as a file moved without changes.

2. Then, *FileB.txt* is handled. There is exactly one matching file in the list of deleted files: *FileE.txt*. So, the heuristic assumes that a move has been performed from *FileE.txt* to *FileB.txt* (i.e. *FileE.txt* has been renamed to *FileB.txt*).

3. Then, *FileC.txt* is checked. It has the same content as *FileB.txt*. Thus, one could say that *FileE.txt* has been renamed to *FileC.txt*. However, *FileE.txt* has already been marked as part of a move. Since a file can only be part of one move, the potential move between files *FileE.txt* and *FileC.txt* is ignored. Because there are no other files matching the content of *FileC.txt*, this file is not classified as a file moved without changes.

4. Finally, *FileD.txt* is handled. There is exactly one deleted file with the same content (*FileF.txt*). Because of this, the heuristic assumes that a move has been performed from *FileF.txt* to *FileD.txt* (i.e. *FileF.txt* has been renamed to *FileD.txt*).

After the moves without changes have been detected, the detection of moves with changes (also called "fuzzy move detection") is performed. For this purpose, all added and deleted text files that have not been classified as moved without changes are compared. If an added or deleted file is either a binary file, or it has been classified as being part of a move without changes, it is not taken

into account in the fuzzy move detection algorithm. In the following, a tuple (A, B), with $A \in AddedFiles$ and $B \in DeletedFiles$, is called "fuzzy move candidate". For the sake of simplicity, we say that for a fuzzy move candidate X, X_1 corresponds to the first entry in the tuple (which is the added file), and X_2 corresponds to the second entry in the tuple (which is the deleted file).

For every possible fuzzy move candidate, the "Similarity Index" of their contents is computed. The Similarity Index is a number between 0 and 1, with 0 indicating that the two compared files are completely different, and 1 indicating that the two files are identical.
The Similarity Index of two file contents $ContentA$ and $ContentB$ is computed using the following formula:

$$SimilarityIndex(ContentA, ContentB) = 1 - \frac{Leven(ContentA, ContentB)}{|ContentA| + |ContentB|}$$

Here, $Leven$ is a function that computes the Levenshtein Distance between two text strings. Konstantinidis (2007) defines the Levenshtein Distance as follows: "The edit distance (or Levenshtein distance) between two words is the smallest number of substitutions, insertions, and deletions of symbols that can be used to transform one of the words into the other." Thus, in this formula, $Leven(ContentA, ContentB)$ returns the number of characters that are different when comparing $ContentA$ and $ContentB$. This number gets normalized by $|ContentA| + |ContentB|$. Here, we say that $|ContentA|$ is the length (i.e. number of characters) of string $ContentA$, and $|ContentB|$ is the length of string $ContentB$.
If $ContentA$ and $ContentB$ are equal, then $Leven(ContentA, ContentB)$ returns 0 and the Similarity Index is 1. If all characters found in $ContentA$ are completely different from the ones found in $ContentB$, then $Leven(ContentA, ContentB)$ is equal to $|ContentA| + |ContentB|$, leading to a Similarity Index of 0. Finally, to avoid division by zero, we say that the Similarity Index of two empty strings (i.e. of two strings with length 0) is 1.

Since the Similarity Index needs to be computed for all pairs of added and deleted files, $\mathcal{O}(nm)$ computations need to be performed, with n being the number of added, and m being the number of deleted files. To reduce the computation time in the implementation, multiple thresholds are defined. For instance, only files with less than a fixed number of characters are taken into account in the comparison. This avoids overly long computation times for single pairs of files. In the implementation, a threshold of $10,000$ characters has been selected, which has shown a good balance between potential performance penalties and quality of results in manual experiments. There is not only a limit for the number of characters per file, but also a hard limit for the total number of files that are compared: If the number of added files multiplied with the number of deleted files is greater than

a fixed threshold, then no fuzzy move detection is performed at all. This number is computed after filtering out files that will not be compared. In the implementation, a threshold of 100 comparisons has been selected, which is again the result of manual evaluation. Finally, only files with a Similarity Index not less than a certain threshold (in the implementation set to 0.5) are considered potential fuzzy moves. This allows to skip the computation of the Similarity Index (and thus the Levenshtein Distance) in some cases: The difference in string lengths of $ContentA$ and $ContentB$ defines a lower bound for the Levenshtein Distance, since it is not possible to change less characters than $||ContentA| - |ContentB||$ to get from $ContentA$ to $ContentB$. Thus, if $1 - \frac{||ContentA| - |ContentB||}{|ContentA| + |ContentB|}$ is less than the defined threshold, then there is no need to compute the actual Similarity Index; The algorithm simply returns 0 for a Similarity Index less than the defined threshold. This is also how a client would most likely expect fuzzy moves to be detected: If two file contents only have a very small fraction of characters in common, then the corresponding files should certainly not be seen as "moved with a lot of changes". Instead, the files should be treated as if they were added and deleted separately.

After the Similarity Indices have been computed for all possible fuzzy move candidates, the fuzzy move candidates are ordered by their Similarity Index, with higher Similarity Indices being listed first. This list is now called S.
The heuristic now uses a "greedy" approach to select which fuzzy moves shall be part of the final result (in the following, the (temporary) list of selected fuzzy move candidates is called T). The algorithm is defined as follows:

1. If S is empty, then stop and return T. Otherwise, select the first item in S and remove it from this list. This item is now called X.
2. Is either X_1 or X_2 already part of T (i.e. is there another fuzzy move candidate Y in T where either $X_1 = Y_1$ or $X_2 = Y_2$)? Then ignore this fuzzy move candidate and go to step 1.
3. Neither X_1 nor X_2 is part of T? Then add X to list T and go to step 1.

In the following, an example is constructed to better illustrate the functionality of the fuzzy move detection algorithm. For this example, two added and two deleted files exist as shown in the following table.

File name	File content
Added files	
FileA.txt	This is a file
FileB.txt	This is file B
Deleted files	
FileC.txt	This is another file
FileD.txt	This is file D

Now, the Similarity Indices for all pairs of these files are computed. This is shown in the following table.

Added file name	Deleted file name	Levenshtein Distance	Similarity Index
FileA.txt	FileC.txt	6	$1 - \frac{6}{14+20} \approx 0.82$
FileA.txt	FileD.txt	4	$1 - \frac{4}{14+14} \approx 0.86$
FileB.txt	FileC.txt	10	$1 - \frac{10}{14+20} \approx 0.71$
FileB.txt	FileD.txt	1	$1 - \frac{1}{14+14} \approx 0.96$

After the Similarity Indices have been computed, the list of fuzzy move candidates is sorted by the Similarity Index as shown in the following table.

Position	Added file name	Deleted file name	Similarity Index
1	FileB.txt	FileD.txt	0.96
2	FileA.txt	FileD.txt	0.86
3	FileA.txt	FileC.txt	0.82
4	FileB.txt	FileC.txt	0.71

Finally, the fuzzy moves can be selected based on this sorted list: The heuristic first handles pair *(FileB.txt, FileD.txt)*. Since no other fuzzy moves have yet been handled, this pair can immediately be added to the final result set. Then, pair *(FileA.txt, FileD.txt)* is handled. Since *FileD.txt* is already part of the final result set, pair *(FileA.txt, FileD.txt)* is skipped. After that, pair *(FileA.txt, FileC.txt)* is handled. Since neither *FileA.txt* nor *FileC.txt* are part of the final result set, pair *(FileA.txt, FileC.txt)* gets added to the final result set. Finally, pair *(FileB.txt, FileC.txt)* is handled. Since *FileB.txt* (and also *FileC.txt*) can already be found in the final result set, pair *(FileB.txt, FileC.txt)* is skipped.
Thus, two pairs of files have been identified as fuzzy moves: According to the algorithm, *FileD.txt* has been renamed to *FileB.txt*, while *FileC.txt* has been renamed to *FileA.txt*.

6.3.3 Result format

As shown before, the `/diff` endpoint distinguishes five different types of changes:

1. Added files
2. Updated files
3. Deleted files
4. Files moved without changes
5. Files moved with changes

When requesting a diff, the backend returns a list of changes, each change classified as one of the five change types. Figure 6.10 shows an example JSON response the server may return: One file (*File1.txt*) has been added, i.e. it does not exist

in the `base` tree but exists in the `target` tree. *File2.txt* has been updated, which means that it exists in the `base` and `target` trees but its content in the `base` tree is different from the content in the `target` tree. *File3.txt* has been classified as moved (i.e. renamed) to *File4.txt* without changes. That means that *File3.txt* exists in the `base` tree but not in the `target` tree, while *File4.txt* exists in the `target` tree but not in the `base` tree.

```
[
  {
    "type": "added",
    "filePath": "/File1.txt",
    "file": {
      "fileMode": "text",
      "content": "This is File 1"
    }
  },
  {
    "type": "updated",
    "filePath": "/File2.txt",
    "originalFile": {
      "fileMode": "text",
      "content": "This is File 2"
    },
    "newFile": {
      "fileMode": "text",
      "content": "This is File 2 with some changes"
    }
  },
  {
    "type": "moved-without-changes",
    "originalFilePath": "/File3.txt",
    "newFilePath": "/File4.txt",
    "file": {
      "fileMode": "text",
      "content": "This is File 3"
    }
  }
]
```

Figure 6.10: An example of a response of the `/diff` endpoint.

7 Evaluation

In this chapter, the final implementation of the Hub VCS is evaluated against the functional and non-functional requirements defined in chapter 3. For this purpose, every requirement is assessed separately to determine whether it has been fulfilled or whether certain aspects are still missing in the final implementation. Since the Contributor Frontend and the Viewer Frontend were barely covered in the previous chapters, this chapter especially highlights selected features of these two frontend applications, together with a description of the relevant endpoints provided by the backend.

7.1 Used notation for endpoints

Since most features of the Hub VCS require interaction with an API endpoint provided by the backend, we define a notation to easily describe an endpoint: For every endpoint, it is essential to know the path the request needs to be performed to, and the HTTP method that needs to be used for this purpose. For example, to retrieve the list of all users stored in the Hub VCS, a client needs to perform an HTTP `GET` request to `(server-url)/users`. In our notation, we write: `GET: /users`. Formally, the notation is using the format `<HTTP Method>: <Path>` with `<HTTP Method>` being the HTTP method (like `GET` or `POST`) the client needs to use when accessing the endpoint, and `<Path>` being the path on the API the request is made to. Optionally, `<Path>` may include parameters which are written using curly braces (e.g. `GET: /repos/{repoId}` to say that the path must be constructed by substituting `{repoId}` with a concrete value, in this case an ID of a repository).

7.2 Evaluation of functional requirements

In this section, the final implementation of the Hub VCS is evaluated against the functional requirements as defined in section 3.4.

Requirement F-1 (Repository Listing)

The API of the Hub VCS provides the endpoint `GET: /repos`. When a client performs a request to this endpoint, a list of all repositories the user has access to is provided. For every entry in this list, the ID and name of the respective repository is provided, as well as information regarding the "owner" of the repository and whether the repository is "private". (A private repository can only be accessed by its owner, which is the user who created this repository.)

Both in the Contributor Frontend and in the Viewer Frontend, the aforementioned endpoint is used to present a list of available repositories. Once a user has selected an entry from this list, he or she is able to explore the respective repository further (when using the Viewer Frontend), or to make changes to it (when using the Contributor Frontend).

In conclusion, since there are no missing aspects from requirement F-1, we say that this requirement has been fulfilled.

Requirement F-2 (Repository Creation)

In order to create a new repository, a client needs to perform a request to the endpoint `POST: /repos`. This endpoint expects a JSON payload in the request body that includes the name of the new repository, as well as a boolean field to describe whether the repository should be private or not. The name of a repository is not unique, so there may be multiple repositories with the same name. Thus, the name of a repository is seen as "meta information" and cannot be used to reference (identify) a particular repository; In order to identify a repository, its ID must be used. Once the repository has successfully been created, the API returns the ID of the created repository.

If a client wants to change the name or "private" flag of a repository, a request to the endpoint `PATCH: /repos/{repoId}` needs to be performed, replacing `{repoId}` with the ID of the repository that shall be updated. The expected JSON payload is the same as in the aforementioned `POST` endpoint. This feature was not explicitly requested in requirement F-2, but it has been identified as a useful addition to improve the user experience. It should be noted that this endpoint is only accessible for the owner of a particular repository.

When a user of the Hub VCS wants to create a new repository or intends to update an existing one, he or she needs to use the Contributor Frontend: Here, the user is able to open a page called "Manage Repositories" where all of the repositories owned by him or her are listed with the possibility to update them, as well as a button "Add Repository" allowing the user to open a dialog to create a new repository.

In conclusion, since there are no missing aspects from requirement F-2, we say that this requirement has been fulfilled.

Requirement F-3 (Repository Deletion)

To delete an existing repository, the owner of this repository needs to perform a request to the endpoint `DELETE: /repos/{repoId}`, replacing `{repoId}` with the ID of the repository that shall be deleted.

In the Contributor Frontend, this functionality can be found on the "Manage Repositories" page that was also used to create or update repositories. On this page, the owner of a repository is able to use a "delete" button which, after asking for a separate confirmation ("Do you really want to delete the repository 'Repository Name'?"), triggers a request to the aforementioned endpoint.

In conclusion, since there are no missing aspects from requirement F-3, we say that this requirement has been fulfilled.

Requirement F-4 (Branch Listing)

To retrieve the list of all branches in a particular repository, a client needs to send a request to the endpoint `GET: /repos/{repoId}/branches`. If the client has proper access permissions (i.e. the repository is either public, or the client is the owner of the private repository), the backend will respond with a list of branches. Every list item contains the ID and the name of the respective branch. It should be noted that, similar to repository names, branch names are not unique and thus cannot be used to identify a particular branch.

In the Contributor Frontend and in the Viewer Frontend, the aforementioned endpoint is accessed in multiple situations. For instance, before being able to edit content in the Contributor Frontend, the user needs to checkout a branch and is therefore presented with a list of branches to choose from. This dialog is shown in figure 7.1. Another example can be seen in the Viewer Frontend: In order to explore the files, folders, and history of the head commit of a given branch, the user first needs to select one of the branches from the presented list as shown in figure 7.2. There are many other situations where the aforementioned endpoint is accessed, since branches are a very important part of the version control workflow in the Hub VCS.

In conclusion, since there are no missing aspects from requirement F-4, we say that this requirement has been fulfilled.

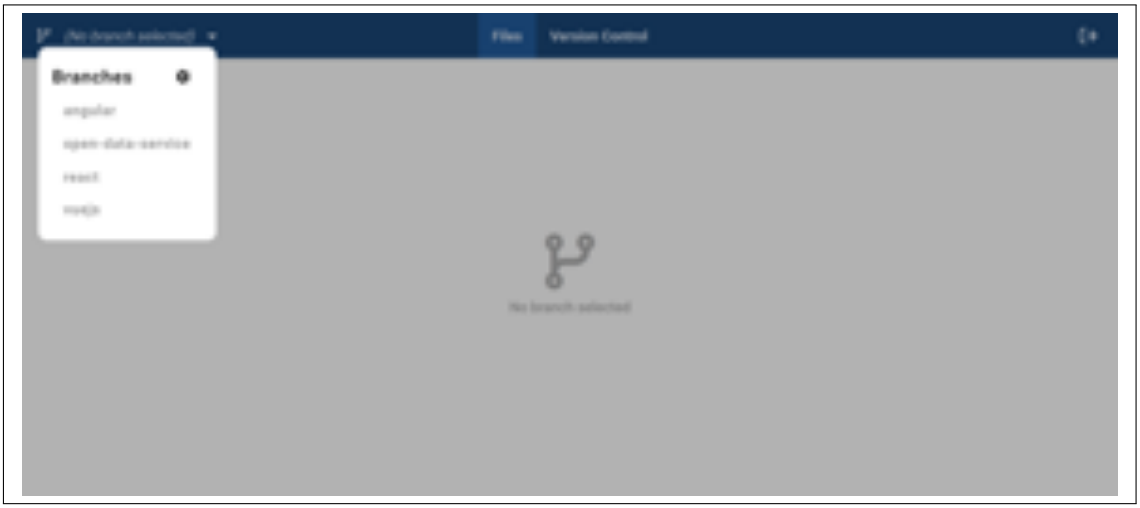


Figure 7.1: Screenshot of the Contributor Frontend showing a branch selector menu. This menu is used to check out a branch in order to make changes to the files and folders of the referenced commit.

Requirement F-5 (Branch Management)

In order to create a new branch in a repository, the client needs to send a request to the `POST: /repos/{repoId}/branches` endpoint. The backend expects the request payload to include the name of the branch that shall be created. Optionally, another branch, tag, or commit, can be specified which will be used as the initial branch head. An example of the request payload needed to create a new branch "my-new-branch", initially pointing to a commit with ID "7", is shown in figure 7.3.

To change the name of a branch, the client needs to perform a request to the `PATCH: /repos/{repoId}/branches/{branchId}` endpoint, replacing `{branchId}` with the ID of the branch that shall be updated[1]. The new name of the branch must be provided in the request payload. Finally, the endpoint `DELETE: /repos/{repoId}/branches/{branchId}` allows clients to delete a given branch.

In the Contributor Frontend, users can open the "Manage Branches" page after selecting a repository and navigating to the "Version Control" page. On the "Manage Branches" page, a list of branches is shown with the possibility to update and delete a particular branch (except for the currently checked out branch). This

[1] At this point, one design decision becomes apparent: Resources, for instance repositories, branches, commits, and the like, are always referenced using their IDs. Thus, the ID is a central element of the Hub VCS and is also in many places shown to the users of the two frontend applications.

Figure 7.2: Screenshot of the Viewer Frontend showing a commit selector menu. This menu is used to select a branch, tag, or commit ID, in order to explore the files, folders, and history of the referenced commit.

```
{
  "name": "my-new-branch",
  "initialHead": {
    "type": "commit",
    "commitId": 7
  }
}
```

Figure 7.3: An example of a request body used to create a new branch with an initial branch head pointing to commit 7.

is shown in figure 7.4. Also, a new branch can be created on this page by providing a branch name and optionally selecting an initial branch head, which is shown in figure 7.5.

In conclusion, since there are no missing aspects from requirement F-5, we say that this requirement has been fulfilled.

Requirement F-6 (Tag Management)

Tags are managed in a similar way as branches. The `GET: /repos/{repoId}/tags` endpoint makes it possible to retrieve a list of all tags for a given repository. A tag has an ID, a name (which again cannot be used as an identifier), and a description. To create a new tag, the

Figure 7.4: Screenshot of the Contributor Frontend showing the "Manage Branches" page.

`POST: /repos/{repoId}/tags` endpoint shall be used, which works similarly to the endpoint used to create branches. It is also possible to update a tag by using the `PATCH: /repos/{repoId}/tags/{tagId}` endpoint, replacing `{tagId}` with the ID of the tag that shall be updated. Finally, a tag can be deleted using the `DELETE: /repos/{repoId}/tags/{tagId}` endpoint.

The most notable difference between a tag and a branch in the Hub VCS is that the commit pointed to cannot be changed for a tag, whereas the head commit of a branch changes over time (by the creation of new commits).

In the Contributor Frontend, the handling of tags is very similar to the handling of branches and is thus not described in more detail here.

In conclusion, since there are no missing aspects from requirement F-6, we say that this requirement has been fulfilled.

Requirement F-7 (Checkout)

In order to perform a checkout, the client needs to fetch the head commit of a given branch. This is possible by using the `GET: /repos/{repoId}/commits/{commitReference}` endpoint. This endpoint allows to fetch the data of a commit either directly by the ID of the commit itself, or by the ID of a branch or tag. This reference to a commit, branch, or tag, is called a "Commit Reference". In the use case of requirement F-7, it is necessary to fetch the head commit of a branch, which is why the client needs to substitute `{commitReference}` with a string in the format

Figure 7.5: Screenshot of the Contributor Frontend showing the "Create Branch" dialog on the "Manage Branches" page.

`branch-x`, with "x" being the ID of the branch the head commit shall be fetched for. For instance, to fetch the head commit of branch 5 in repository 3, the request needs to be performed to `/repos/3/commits/branch-5`. The response from the backend includes the ID of the fetched commit, meta information like the commit title and author, and the entire file/folder tree. An example of a response is shown in figure 7.6. It should be noted that this endpoint is used in various places: It is not only used to perform a checkout, but also, for instance, in the Viewer Frontend to allow users to explore the files and folders of a given commit, tag, or branch.

In the Contributor Frontend, the user performs a checkout by selecting a branch as already shown in the evaluation of requirement F-4: After the branch has been selected, a request to the aforementioned endpoint is made to fetch the details (and most importantly the file/folder tree) of the head commit of the selected branch. The user is then able to explore the files and folders, make changes to the file/folder tree, to create new commits, and more.

In conclusion, since there are no missing aspects from requirement F-7, we say that this requirement has been fulfilled.

Requirements F-8 (Edit) and F-9 (File and Folder Management)

These two requirements are evaluated together, since they cover a very similar subject. In the Contributor Frontend, after a checkout has been performed, the

```
{
  "id": 8,
  "creationDate": 1640991600,
  "title": "Add important changes",
  "description": "This commit introduces an important update
which resolves multiple issues discussed with the team.",
  "author": {
    "id": 1,
    "name": "Sample User"
  },
  "parents": [7],
  "tree": {
    "children": {
      "files": {
        "test.txt": {
          "fileMode": "text",
          "content": "this is a file"
        },
        "test2.txt": {
          "fileMode": "text",
          "content": "this is another file"
        }
      },
      "folders": {}
    }
  }
}
```

Figure 7.6: An example of a response from the backend when requesting the details of a commit.

user is able to explore files and folders on the “Files” page. A screenshot of this page is shown in figure 7.7: On the “Files” page, a file/folder tree is shown, which not only enables the user to get an overview of the files and folders present in the checked out commit, but also provides features to modify this tree. For instance, it is possible to create new files and folders, to delete existing ones, and also to move single files or entire folders. The “move” feature, as shown in figure 7.8, has been designed in a way that it also allows to rename the respective file or folder, which is why there is no separate “rename” option. When selecting a file, an editor is shown which either allows to edit a text file (as shown in figure 7.7), or to upload a binary file (as shown in figure 7.9). Thus, the “Files” page is essentially an in-browser file manager, combined with a file editor. It should be noted that all of the aforementioned operations are performed fully on the client side (i.e. in the browser of the user) - the changes made to the file/folder tree are only transferred to the backend once the user decides to create a commit.

In conclusion, since there are no missing aspects from requirements F-8 and F-9, we say that these requirement have been fulfilled.

Figure 7.7: Screenshot of the Contributor Frontend showing a text file editor on the “Files” page.

Requirement F-10 (Status and Revert)

Similar to requirements F-8 and F-9, requirement F-10 has been implemented in the Contributor Frontend in a way that it does not require communication with the backend. On the “Current Changes” page of the “Version Control” section, a list of changes is shown. This list includes the added, updated, and deleted files, in comparison to the state the file/folder tree was in on the last checkout.

Figure 7.8: Screenshot of the Contributor Frontend showing the "Move File" dialog on the "Files" page. This dialog not only makes it possible to move the selected file to a new directory, but also to rename this file.

Thus, this list includes all changes the user has made on the "Files" page. It is possible to "preview" a change (i.e. to see the diff compared to the checked out file/folder tree) as shown in figure 7.10. Also, it is possible to revert the changes made in individual files: If a file has been deleted, it is possible to restore it. If a file has been added, it is possible to delete it. And if a file has been updated, it is possible to undo these modifications.

In conclusion, since there are no missing aspects from requirement F-10, we say that this requirement has been fulfilled.

Requirement F-11 (Commit)

The backend implementation regarding the creation of regular commits has already been covered in chapter 6, which is why this part is not described here in more detail. In the Contributor Frontend, commits are created on the "Current Changes" page: As mentioned before, this page shows a list of the changes that have been performed to the working copy. In order to make one of the changed files part of the next commit, the user selects it in the list of changed files. Once all of the files that shall be part of the commit are selected, and the user has entered a title and description for the commit, he or she can submit the commit to the backend. It should be noted that it is not necessary to select every single changed file to be part of one commit - if a file has not been selected, then it remains unchanged and can, for instance, be made part of a future commit. This allows users to make large changes to their working copy and then to "split" these changes into

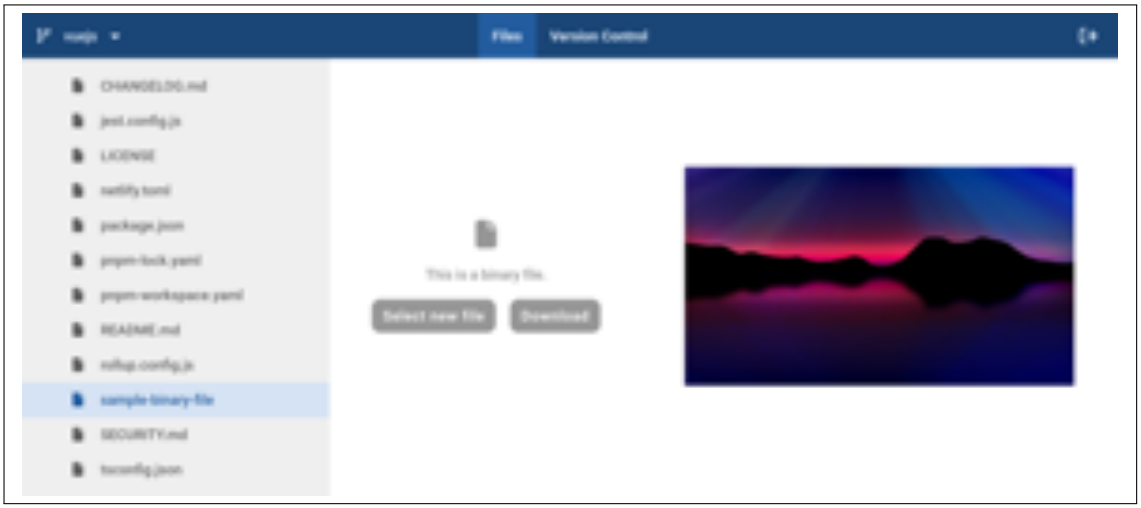


Figure 7.9: Screenshot of the Contributor Frontend showing a preview of a binary file on the "Files" page.

multiple small commits, instead of having to create one large commit.

In conclusion, since there are no missing aspects from requirement F-11, we say that this requirement has been fulfilled.

Requirement F-12 (Update)

When two users are working on the same branch, and one user pushes changes to this branch, then the other user needs to perform an update of the working copy before being able to create a commit. For this purpose, the "Update" page has been implemented in the Contributor Frontend: If the working copy is not up-to-date, a message on this page is shown to indicate that an update needs to be performed. (A similar message is also shown when trying to create a commit.) To update the working copy, the "Update" page presents a "three-way-diff" view: Here, three trees (the originally checked out tree, the working copy, and the tree belonging to the current head of the branch as fetched from the backend) are compared. Every file that is not equal in the three trees is shown to the user. The goal of this three-way-diff view is to build a new working copy that is based on the current head of the branch and still includes the changes the user has previously made to the working copy. To do that, the user is presented with two diffs for every single file as shown in figure 7.11: On the left side, the state of the file in the working copy is shown, while visualizing the changes compared to the originally checked out file. On the right, the state of the file in the "incoming" tree (i.e. of the current head of the branch) is shown, again while visualizing the changes compared to the file in the "base tree". Now, the user needs to select one

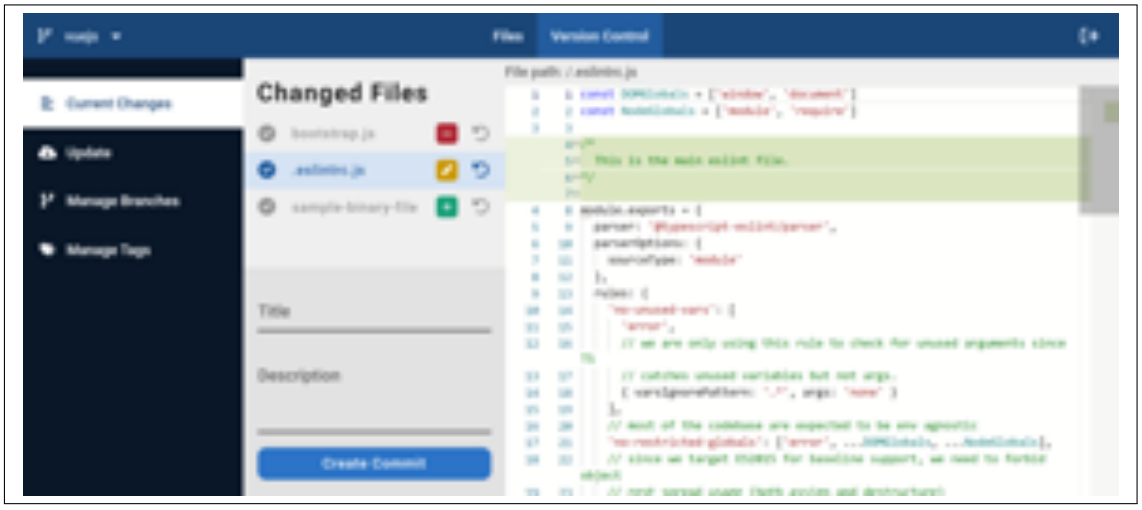


Figure 7.10: Screenshot of the Contributor Frontend showing the "Current Changes" page.

of these changes to be part of the new working copy. This "result file" is shown in the middle column. As shown in figures 7.12 and 7.13, it is possible for the user to edit the file shown in the middle, e.g. in order to manually merge more complex changes made in the files shown on the left and right. Once the user has finished working on one particular file, he or she needs to mark this conflict as "resolved". Once all conflicts have been resolved, the changes are applied to the working copy and the user is now able to continue the work on an up-to-date working copy.

As already mentioned, the three-way-diff view allows the user to manually inspect every single changed file and to apply changes to these files in order to define how they should be merged. However, the user does not have to take a look at every single file when performing this merge: The three-way-diff view is able to perform "smart guesses" to automatically select the content of the result file in certain situations. For instance, if a particular file has not been modified in the working copy, but it has been modified in the incoming tree, then the latter is automatically selected as the result file, and the user does not have to perform any actions for this file. Or, if a file has been deleted in the working copy, but has not been changed in the incoming tree, then this file is automatically marked for deletion. Of course, the user is able to override these "smart guesses" - they are merely used to assist the user especially in cases where a large number of files needs to be handled in the update.

In conclusion, since there are no missing aspects from requirement F-12, we say that this requirement has been fulfilled.

Figure 7.11: Screenshot of the Contributor Frontend showing the three-way-diff view of the "Update" page. In this example scenario, three text editors are shown in order to handle the changes made in "file1.txt".

Requirement F-13 (Merge)

In order to perform a merge, the user of the Contributor Frontend needs to use the "Merge Branches" feature found on the "Manage Branches" page. To perform a merge, the user first needs to select the source branch and the target branch of this merge operation. The Contributor Frontend then loads the "merge base", which is a common ancestor of the two commits the source and target branches point to. To load this merge base, the Contributor Frontend performs a request to the `GET: /repos/{repoId}/commits/merge-base` endpoint. This endpoint expects the two commit IDs to be passed as query parameters. For instance, to request the merge base of commits 5 and 8 in repository 3, a request to `/repos/3/commits/merge-base?commits[]=5&commits[]=8` needs to be performed. The response of this endpoint includes the ID of the merge base, as well as the entire file/folder tree.

There are multiple situations in which a merge cannot be performed. For instance, if the target branch is "ahead" of the source branch, i.e. if the head commit of the source branch is an ancestor of the head commit of the target branch, a merge is rejected. Or, if the two commits do not have a common merge base, then a merge is also not possible. All of these cases are handled in the Contributor Frontend by showing an informative error message and aborting the merge process.

If the merge base could successfully be loaded, the same three-way-diff view is shown as already used for the update of the working copy. Finally, the merge commit can be created, which internally uses the same endpoint as used for the

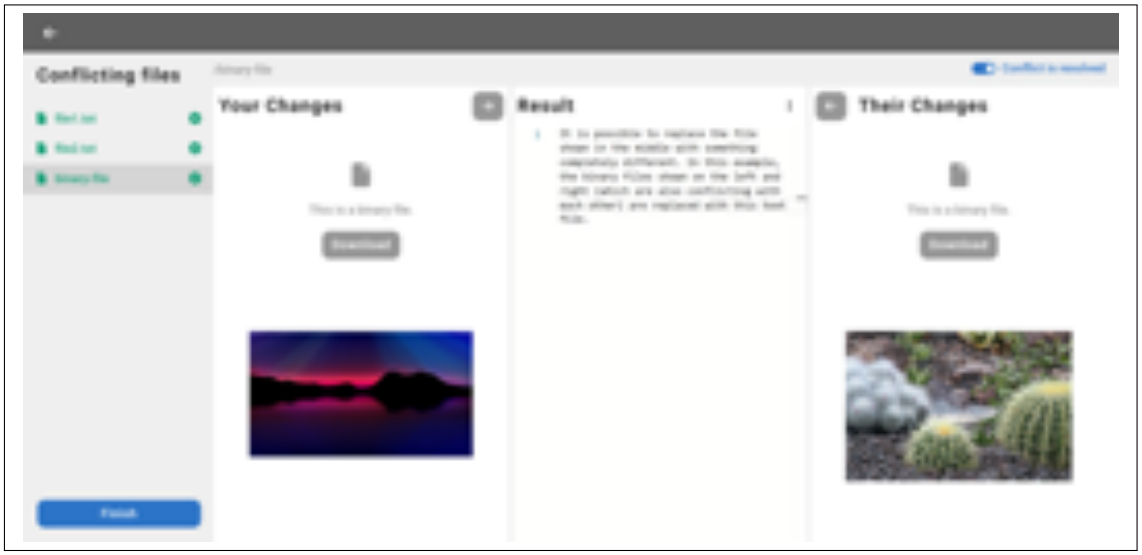


Figure 7.12: Screenshot of the Contributor Frontend showing the three-way-diff view of the "Update" page. In this example scenario, two conflicting binary files are shown, and the user has decided to replace the binary files with a text file.

creation of a regular commit. The only difference in the request payload compared to the creation of a regular commit is that there is another field `mergeCommitId`, which indicates that a merge with this commit ID shall be performed.

In conclusion, since there are no missing aspects from requirement F-13, we say that this requirement has been fulfilled.

Requirement F-14 (History)

To retrieve the history of a commit (i.e. the list of ancestors of this commit), the client needs to send a request to the `GET: /repos/{repoId}/commits/{commitReference}/history` endpoint. The response of this endpoint contains a list of commits, which are the ancestors of the referenced commit as well as the referenced commit itself. Every entry in this list contains the commit ID, the IDs of the commit parents, as well as meta information like the commit title, author, and more. It is also possible to fetch the history of a file or folder by specifying a query parameter called "path". In this case, only commits that introduced, updated, or deleted a given file or folder are returned.

In the Viewer Frontend, the user is able to explore the commit history after selecting a commit, tag, or branch, on the "Files" page as shown in figure 7.14. When selecting a file or folder, the history of the selected item is shown, while

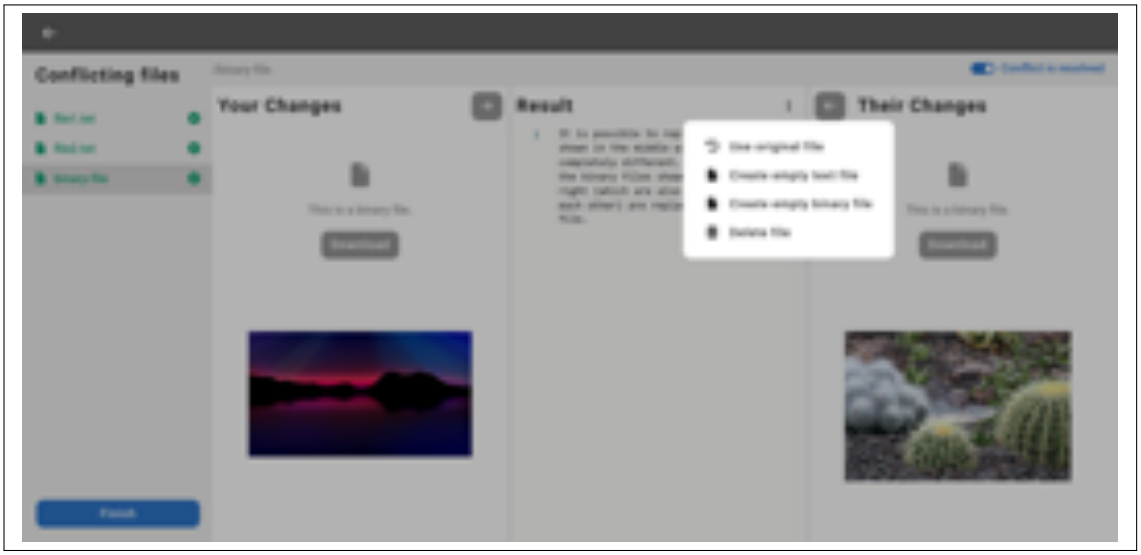


Figure 7.13: Screenshot of the Contributor Frontend showing the three-way-diff view of the "Update" page. In this example scenario, two conflicting binary files are shown, and the user has opened a menu that, among others, allows to replace the "result file" with an empty text file or to delete this file.

the entire commit history is shown when selecting the name of the repository.

In conclusion, since there are no missing aspects from requirement F-14, we say that this requirement has been fulfilled.

Requirement F-15 (Diff)

The backend implementation regarding the retrieval of the diff between two commits has already been covered in chapter 6, which is why this part is not described here in more detail. In the Viewer Frontend, the user is able to compare two arbitrary commits on the "Diff" page: After selecting the "base" and the "target" commit, the list of added, updated, deleted, and moved files is shown. When selecting an item from this list, the file content is displayed, which makes it possible to explore the respective file in more detail. If a file has been updated or moved with changes, the changes introduced in the target commit compared to the base commit are highlighted as shown in figure 7.15.

In conclusion, since there are no missing aspects from requirement F-15, we say that this requirement has been fulfilled.

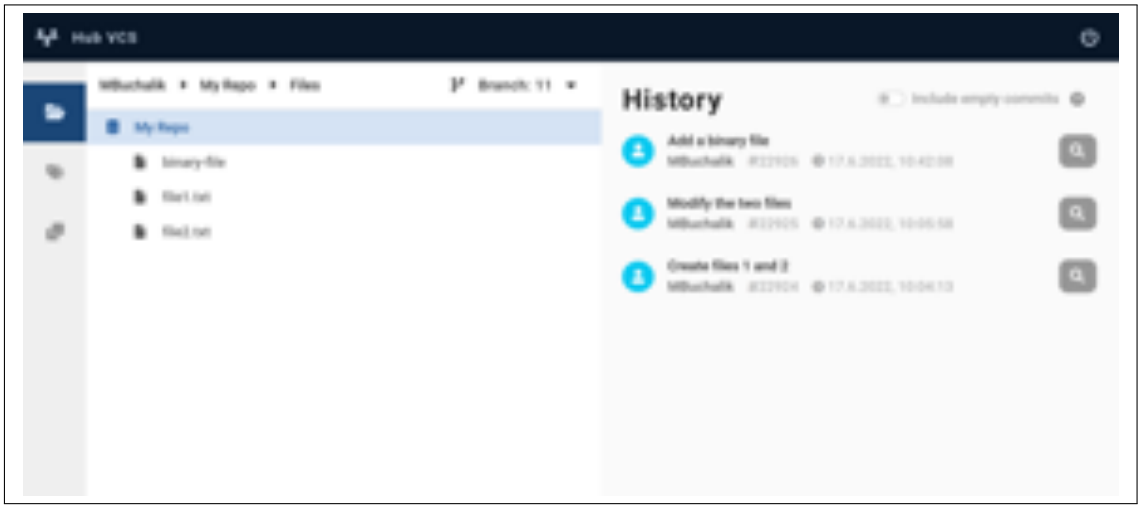


Figure 7.14: Screenshot of the Viewer Frontend showing the commit history of a selected branch.

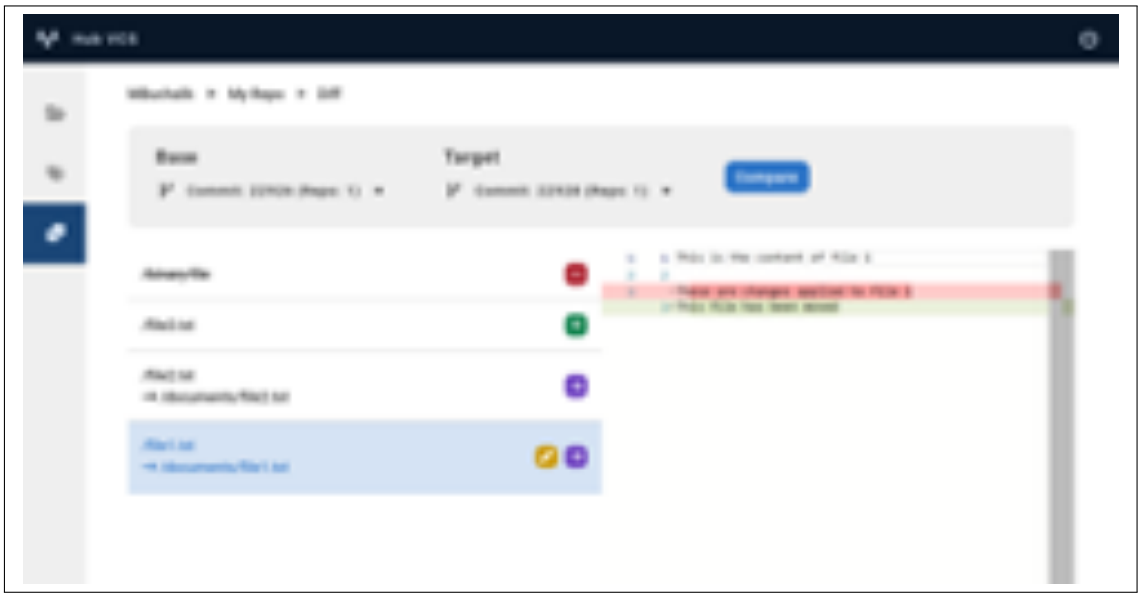


Figure 7.15: Screenshot of the Viewer Frontend showing the "Diff" page.

7.3 Evaluation of non-functional requirements

In this section, the final implementation of the Hub VCS is evaluated against the non-functional requirements as defined in section 3.5.

Requirement N-1 (Architecture)

The clients can communicate with the backend of the Hub VCS using the HTTP API. The two frontend applications are both web-based and thus in compliance with requirement N-1. There are no missing aspects from requirement N-1, thus we say that this requirement has been fulfilled.

Requirement N-2 (Programming Languages and Frameworks)

As described in chapter 5, not only the backend and frontend applications, but also all other components like the Importer, have been implemented using TypeScript. Additionally, the two frontend applications have been built using Vue.js, making it possible to reuse pieces of the applications in other parts of the JValue Project.

Unfortunately, after most features of the Contributor Frontend and Viewer Frontend had been implemented, the development of some applications in the JValue Project started to be focused around the usage of React[2] instead of Vue.js. In those projects, it is certainly not trivial to reuse the Vue.js components built for the Hub VCS. But thanks to the Shared package, which only requires TypeScript to be available in a project, it is quite easily possible to access all features of the Hub VCS in a React project (or any other kind of project based on TypeScript). Thus, one could now see the Contributor Frontend and Viewer Frontend as "reference implementations" that not only showcase all features of the Hub VCS, but also provide many recommendations regarding UI and UX design.

In conclusion, since there are no missing aspects from requirement N-2, we say that this requirement has been fulfilled.

Requirement N-3 (Code Style)

All components of the Hub VCS follow one central set of linter rules defined using ESLint[3]. The rules are based on the ones used across other applications in the

[2]https://reactjs.org, accessed on June 21, 2022
[3]https://eslint.org, accessed on June 21, 2022

JValue Project like the DEWB, which makes the code follow one common style. For the Hub VCS, these rules have been extended to be even stricter than the original ones in order to further improve the quality of the implementation.

Hence, we say that requirement N-3 has been fulfilled.

Requirement N-4 (Testing)

As already described in chapter 5, the endpoints of the backend of the Hub VCS are tested using system tests. For every single endpoint, at least one test case has been defined in order to ensure that all endpoints are working according to their specification, and to make sure that edge cases are properly handled. Additionally, the performance of the Hub VCS has been manually tested after importing multiple Git repositories using the Importer.

Hence, we say that requirement N-4 has been fulfilled.

Requirement N-5 (API Documentation)

Every single HTTP API endpoint has been documented in a text file, describing which features the respective endpoint provides, which payload it expects, which data the response contains, and more. Additionally, every endpoint is also defined using a TypeScript object, which contains all relevant information needed to perform and validate a request and its response. These objects, which have been called "Endpoint Descriptions", are not only used by the clients to make requests and validate the responses, but also by the backend to verify e.g. whether the payload of a request has the correct structure. All Endpoint Descriptions are part of the Shared package, which enables developers to reuse them in other applications that shall communicate with the backend of the Hub VCS.

In conclusion, since there are no missing aspects from requirement N-5, we say that this requirement has been fulfilled.

7.4 Summary

As described in section 3.3, to summarize by which degree the functional and non-functional requirements have been fulfilled, the fraction $\frac{\#\text{CompletedRequirements}}{\#\text{TotalRequirements}}$ can be computed.

In total, there are 15 functional requirements. As shown in section 7.2, all required features have been implemented for every single functional requirement. Thus, all 15 functional requirements can be classified as "completed", which means that $\frac{15}{15} = 100\%$ of the functional requirements have been fulfilled by the final system. This shows that the final system covers every necessary feature that has been identified in section 3.4.

The same metric can also be computed for the non-functional requirements: In total, there are five non-functional requirements. As shown in section 7.3, every single non-functional requirement has been met by the final system. Thus, all five non-functional requirements can be classified as "completed", meaning that $\frac{5}{5} = 100\%$ of the non-functional requirements have been fulfilled by the final system. Thus, the Hub VCS has been designed in full accordance with the non-functional requirements defined in section 3.5.

8 Conclusion

This thesis has contributed to the JValue Project by demonstrating how a VCS tailored to the technical and logical structure of the DEWB, JValue Hub, and similar applications, could be designed and implemented. For this purpose, the most important features of a VCS (the Core Operations) have been identified, which constitute the basis of any general VCS. Based on these Core Operations and the context of the JValue Project, the functional and non-functional requirements have been defined, which were not only used as a "framework" for the research and development needed for the Hub VCS, but also allowed to evaluate whether the finished implementation matches the initial expectations regarding a (good) solution. Based on these requirements, a theoretical foundation has been established that covers the most relevant aspects that need to be considered before implementing a VCS in the context of the JValue Project. This theoretical foundation is independent of a concrete selection of programming languages and the like, which makes it theoretically possible to build upon this foundation in the (distant) future, even if the selection of used technologies in the JValue Project changes. After this theoretical concept had been defined, its feasibility got demonstrated with the implementation of the Hub VCS. For this implementation, not only the theoretical concept, but also the functional and non-functional requirements have been considered, in order to establish a system architecture that fits in the context of the JValue Project. Finally, the finished system has been evaluated against the functional and non-functional requirements. All requirements have been met by the Hub VCS, which is a strong indicator that the Hub VCS could indeed provide a good extension to the set of applications already present in the context of the JValue Project.

In conclusion, this thesis has provided important insights that will influence the future developments in the context of the JValue Project, supporting the JValue Project on its mission to "make open data easy, safe, and reliable".

Appendix

Mockups of DEWB and JValue Hub

At the time of writing this thesis, the DEWB and the JValue Hub were in an early stage of development. However, multiple visions both for the DEWB and the JValue Hub were present in the form of Mockups. The final products will most likely differ in major aspects from these Mockups. Still, these images are able to transport the general ideas behind the DEWB and the JValue Hub. Thus, in the following, Mockups are presented showing early design concepts regarding the DEWB and the JValue Hub.

Figure 1: A configuration step in the DEWB. On the left, the user is able to define the expected "schema" of the files present in a data source. In this particular example, the user is modelling the schema of a tabular file. Once the user makes a change on the left side, feedback is immediately presented on the right side: Here, a preview of the tabular file is presented, which gets evaluated against the provided schema. Mismatches between the configuration and the loaded file are highlighted to allow the user to quickly identify problems with the provided configuration.

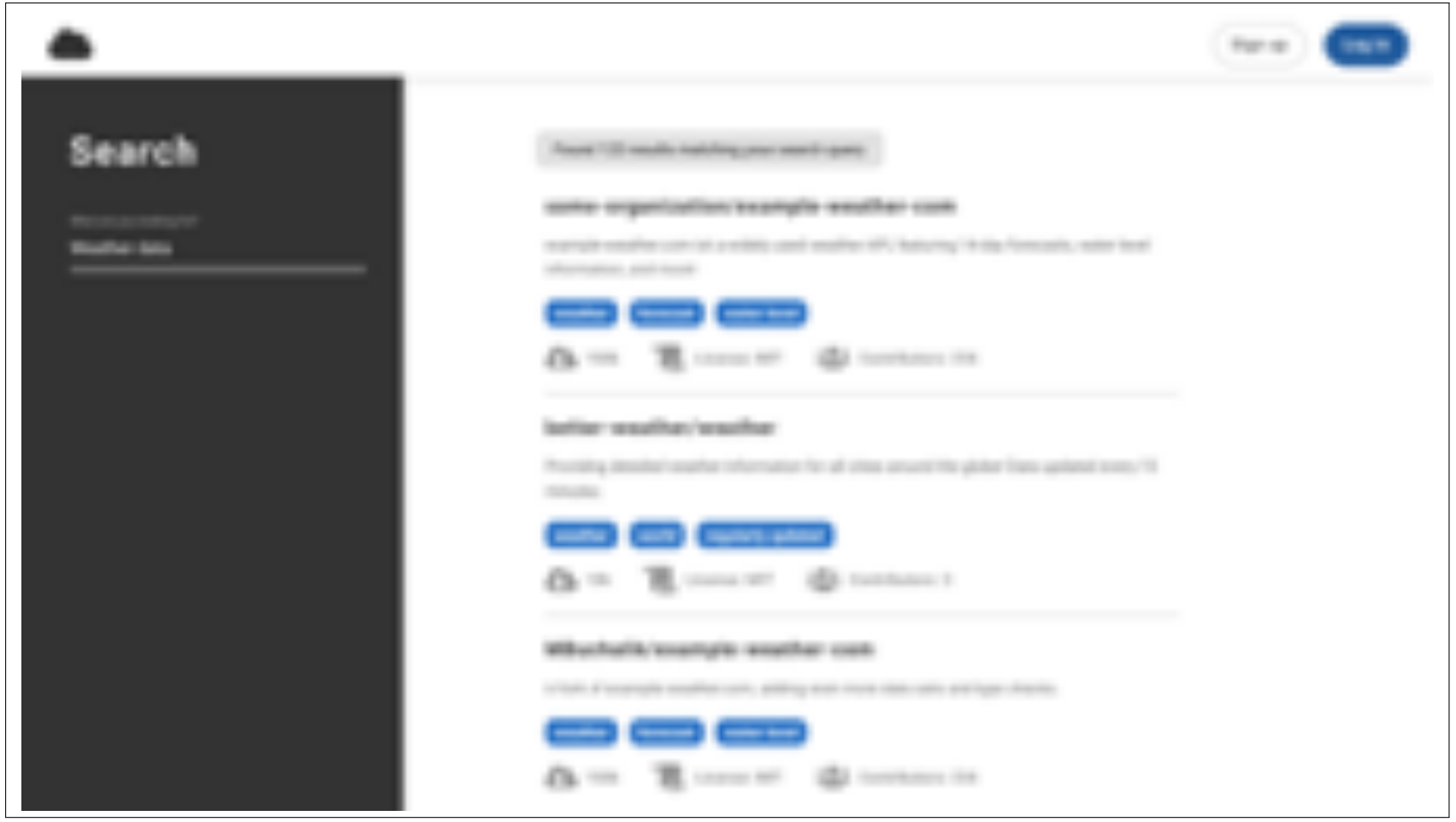


Figure 2: One important feature of the JValue Hub is related to the exploration of existing repositories: If a user wants to access a particular kind of data source (e.g. one providing weather data), he or she should be able to simply enter corresponding search terms and get a list of all matching repositories, including license information, information regarding how many times data from this repository has been accessed, and more.

Figure 3: After the user has selected a repository in the JValue Hub search view as shown in figure 2, more details about this repository are presented. Here, a textual description ("README file") is shown, as well as keywords, license information, and further options to explore this particular repository.

References

Chacon, S. & Straub, B. (2022). Pro git. (Vol. 2.1.338-2-g8a81047, 2022-04-05, pp. 12–13, 63–64, 432–442). Apress Berkeley, CA.

de Alwis, B. & Sillito, J. (2009). Why are software projects moving from centralized to decentralized version control systems? (pp. 36–39). doi:10.1109/CHASE.2009.5071408

Haerder, T. & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*(4), 287–317. doi:10.1145/289.291

Huijboom, N. & van den Broek, T. (2011). Open data: An international comparison of strategies. *European Journal of EPractice*, *12*, 1–13.

Kitchin, R. (2014). The data revolution: Big data, open data, data infrastructures & their consequences. (p. xv). SAGE.

Koc, A. & Tansel, A. U. (2011). A survey of version control systems. *ICEME 2011*.

Konstantinidis, S. (2007). Computing the edit distance of a regular language. *Information and Computation*, *205*(9), 1307–1316.

Loeliger, J. & McCullough, M. (2012). Version control with git: Powerful tools and techniques for collaborative software development. (p. 1). O'Reilly Media.

Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M. & Brinkkemper, S. (2016). Improving agile requirements: The quality user story framework and tool. *Requirements Engineering*. doi:10.1007/s00766-016-0250-x

Mehlhorn, K. & Sanders, P. (2008). *Algorithms and data structures: The basic toolbox*. Springer.

Merkle, R. C. (1987). A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques* (pp. 369–378). Springer.

Merkle, R. C. (1989a). A certified digital signature. In *Conference on the theory and application of cryptology* (pp. 218–238). Springer.

Merkle, R. C. (1989b). One way hash functions and DES. In *Conference on the theory and application of cryptology* (pp. 428–446). Springer.

Niaz, M. S. & Saake, G. (2015). Merkle hash tree based techniques for data integrity of outsourced data. *GvD*, *1366*, 66–71.

Otte, S. (2009). Version control systems. Computer Systems and Telematics Working Group.

Rivero, J. M., Grigera, J., Rossi, G., Luna, E. R., Montero, F. & Gaedke, M. (2014). Mockup-driven development: Providing agile support for model-driven web engineering. *Information and Software Technology*, *56*(6), 670–687.

Sink, E. (2011). *Version control by example*. Pyrenean Gold Press Champaign, IL.

Szydlo, M. (2004). Merkle tree traversal in log space and time. In *International conference on the theory and applications of cryptographic techniques* (pp. 541–554). Springer.

Vaidya, S., Torres-Arias, S., Curtmola, R. & Cappos, J. (2019). Commit signatures for centralized version control systems. In *Ifip international conference on ict systems security and privacy protection* (pp. 359–373). Springer.

Wessels, B., Finn, R., Wadhwa, K. & Sveinsdottir, T. (2017). *Open data and the knowledge society*. Amsterdam University Press. doi:10.1515/9789048529360