TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Tavo Annus 186060IAIB
Marten Jürgenson 185045IAIB

DEVELOPMENT OF TIME TRACKING AND VISUALIZATION SYSTEM FOR PROGRAMMING

Bachelor Thesis

Supervisor
Ago Luberg
M Sc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tavo Annus 186060IAIB

Marten Jürgenson 185045IAIB

PROGRAMMEERIMISELE KULUVA AJA JÄLGIMISE JA VISUALISEERIMISE SÜSTEEMI ARENDAMINE

Bakalaureusetöö

Juhendaja

Ago Luberg

M Sc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author:	Tavo Annus, Marten Jürgenson	.... (signature)
Date:	May 18th, 2021	

Annotatsioon

Käesoleva bakalaureusetöö raames on tehtud Tallinna Tehnikaülikooli jaoks vabavaraline programmeerimise aja jälgimise ja visualiseerimise süsteem. Eesmärk on anda nii õppejõule kui ka tudengitele ülevaade, millistele ülesannetele kõige rohkem aega kulub ning kuidas on ajaline panus jaotunud.

Süsteem baseerub rakendusel Git-Time-Metric, mille ümber ehitati üles ülejäänud süsteem. Nüüd koosneb süsteem viiest erinevast rakendusest mis on jaotatud ka viite erinevasse kihti. Kaks nendes paiknevad kliendi masinas ning ülejäänud kolm serveris. Rakendustes on kasutatud Rust, Go, TypeScript ning Kotlin programmeerimise keeli, PostgreSQL andmebaasi.

Töö tulemusena täiendati Git-Time-Metric rakendust, loodi Jetbrains'i rakendustega ühildumiseks *plugin*, loodi *Debian package*'sse pakendatud rakendus andmete lugemiseks paljudest Git repositooriumitest ning valmistati *backend* ja *frontend* rakendused andmete mugaval kujul hoidmiseks ning visualiseerimiseks.

Valminud rakenduse veebiliides on kättesaadav aadressil https://cs.ttu.ee/services/gtm/front ning ülejäänud osad on leitavad GitHub'is https://github.com/DEVELOPEST.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 46 leheküljel, 7 peatükki, 19 joonist, 14 tabelit.

Abstract

As part of this thesis, open source time tracking system was built for Tallinn University of Technology. The purpose of the system is to give an overview of the time put into each task and how is this effort is distributed over time. The system is built for both lecturers and students, and it is developed in a way that it could be also used for personal projects or companies.

The system is built around an open source app called Git-Time-Metric. Now the system consists of five different applications distributed into five layers. Two of the layers are on a client machine and three are hosted on a server. Rust, Go, TypeScript and Kotlin programming languages were used in writing the applications.

As a result of this work, Git-Time-Metric app was improved, a plugin was built for Jetbrains' editors, a Debian package was built for the app responsible for collecting data from multiple Git repositories and a frontend - backend combo was built for storing and visualizing data.

Deployed web app is accessible at https://cs.ttu.ee/services/gtm/front and rest of the applications can be found on GitHub https://github.com/DEVELOPEST.

The thesis is in English and contains 46 pages of text, 7 chapters, 19 figures, 14 tables.

List of abbreviations and terms

API	Application Programming Interface
BSD-3	Berkeley Software Distribution 3 Clause Licence
CI/CD	Continuous Integration and Continuous Development
CLI	Command Line Interface
CSV	Comma Separated Values
DDD	Domain Driven Design
ERD	Entity Relationship Diagram
GPLv3	GNU General Public License v3.0
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
JSON	Java Script Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
LGPLv3	GNU Lesser General Public License v3.0
MIT	Massachusetts Institute of Technology
ORM	Object-relational mapping
REST	Representational State Transfer
SQL	Structured Query Language
SSH	Secure Shell Protocol
VCS	Version Control System

Table of Contents

List of Figures

List of Tables

1. Introduction

The time spent on a programming task can be divided into two parts: time spent on writing code and supportive activities. As the time spent on writing actual code directly contributes towards a solution, but time spent on supportive activities does not, many companies seek to analyse this kind of data.

For working people, time spent on both writing code and all supportive activities will add up to total working hours. As the total consists of only two components, if we manage to measure one we can easily calculate the other. Measuring time on supportive activities is rather hard as they vary a lot in their nature. Different time management tools can be used, but they usually rely on manual user input making them impractical. Therefore, it is easiest to measure time spent on writing code.

Similarly, students time put into completing practical programming subject at university can be divided into three: time spent in lectures/labs, time spent on doing research on your own and time spent on writing actual programs/solutions. Time spent in lectures and labs can be easily measured as they have strict time schedules. Similarly to supportive activities at work, measuring time spent on research has to rely on manual user input as research can be done in various ways some not involving computers at all. Once again, an application for measuring time spent on writing code is needed.

As both companies and universities measure developers performance by measuring the amount tasks completed, or the amount of code written it is also important to measure developer's efficiency in writing code. Measuring statistics such as problems solved is hard as problems vary a lot, and their difficulty is complex to measure. To simplify measuring developer efficiency we can look at the amount of code written as more code written usually results in more work done.

The amount of code written is usually measured in lines added and lines removed as the number of newlines usually describes the amount of logical statements or functions used. To have access statistics such as lines added and lines removed, some kind of access to a version control system is needed.

The goal of this project is to develop an application capable of both measuring time spent on writing code and also displaying it in a readable format to a user. The application must work with a minimal amount of manual input as manual input both discourages people to use it and may produce unrealistic results. People simply may give input based on their feelings, rather than actions.

1.1 Project scope

The scope of this project is to build an application that solves the problem of measuring the time spent on writing programs for TalTech. Although the application is developed for TalTech it should be possible to set it up for tracking time on personal or work projects.

The scope of this project consists of three main parts. The goal of the first phase is to have a working app to track and save programmers time. The second phase focuses on collecting the data together and storing it in a more accessible format. The third phase is to visualize collected data to both lecturers and students.

As most TalTech subjects use Git as a version control system, the application developed must be able to work with projects that use Git as their version control system.

It was discussed with the client that research needs to be done on already existing solutions for the problem and if possible, the project should extend the already existing solution rather than building a new one. It was also discussed that at least parts of the application that are installed on a client machine should be made open source.

2. Project description

Prior to the actual development, the general project framework was set up:

- The roles and responsibilities were split.
- The rules and agreements on developing methodology were made.
- Environments were set up

It was agreed that Tavo Annus is responsible for DevOps and client-side software. Marten Jürgenson is responsible for the backend and frontend development. As the client-side software has less work to do, Tavo will be also working on both the backend and frontend on the latter half of the project.

For project management, GitHub was chosen as it is more common for users to browse projects on GitHub rather than Gitlab. Universities personal Gitlab server was ruled out as it does not allow non-university students to easily access open source parts of our application.

For the communication channel, Discord was used for both text and sound/video communication. The main reason was that we were both used to using Discord, it is free, and it allows reasonably well-formatted code pastes / pinning.

Meetings with TalTech mentor were organized weekly on Teams to have continuous feedback on the project.

2.1 Use cases

To have a more clear understanding of what are the problems for our potential users and how we can solve them, three main use cases were discussed. These use cases are:

1. Time tracking for universities - measuring students' work on programming subjects.
2. Time tracking for companies - using time tracking at work to maximise developers' efficiency.
3. Time tracking for personal usage.

For the use cases, the main benefits and concerns were discussed.

2.1.1 Time tracking for universities

For universities the needs for time tracking are:

- It has to be possible to have an overview of numerous students.
- The time tracking shall not be easily tricked to display bigger / smaller numbers.
- Application has to work on all commonly used operating systems (Windows 10, macOS and Linux).
- Application has to be easy to use and work with minimal user input.
- Application shall have a clear purpose and access rights.

Possibility to have an overview of a group of students is needed to get feedback on subjects. The only sensible way to have an overview of a volume of study in a subject is to measure it on multiple students. This also leads us to a requirement, that it shall not be easy for students to distort their time data. Students commonly like to magnify their effort put into subjects and if it is easy to do, we have no idea if the data is distorted or not. As there is a big number of students in some subjects and students have multiple subjects to attend it has to be easy to install the app. There simply is no time to waste on helping all the students with more complex manual installation. As universities have no right to enforce usage of some operating system, the app also has to work on all commonly used platforms. Lastly, it has to be clear for students, what the application is doing. Otherwise, they may be afraid of the invasion of privacy and not install the application.

2.1.2 Time tracking for companies

For companies the needs for time tracking are:

- It needs to be clear, that no source code is leaked outside the company.
- The time tracking application has to work with repositories containing 10s thousands of commits.
- Time tracking data has to be persisted through complex Git operations such as rebasing.
- It has to be possible to filter time by issue.
- It has to be possible to filter time data in a single repository by the user.
- It shall be possible to dynamically group different repositories.

For companies, more security concerns arise. It is very important to protect source code, therefore they highly prefer the code to never leave their own hardware/servers. As companies have numerous developers working on the same project simultaneously for a long period of time the application has to remain working efficiently when used in repositories containing a large number of commits. Furthermore, as there are many people working on the same repository the time data has to persist through all Git operations. Otherwise, it will start limiting developers, and they will not use it. To have a clear overview of both the work done by user and work done on a certain issue, it has to be possible to join time data over multiple repositories and filter it by users and issues.

2.1.3 Time tracking for personal use

For personal usage, the main requirements for time tracking are:

- Application has to work on all commonly used operating systems.
- Application has to be easy to install / use.
- Application has to be low cost, preferably free.
- Application needs to work with commonly used Git servers (Github, Gitlab.com)

For personal usage, similarly to universities, the application has to work on all commonly used operating systems as otherwise we would by greatly limiting the number of potential users. As there is usually no one to help, the application installation has to be simple enough people with basic computer usage skills can do it. One of the most important requirements for personal use is that the application has to be either very low cost or completely free. Since people work on their personal projects only as much as they like, and they do it inconsistently they are not interested in paying for an app they might not even use. As different developers have personal preferences for which Git server to use, we have to support all the most common ones. As of the beginning of 2021, they are Github.com and Gitlab.com.

2.2 Existing solutions

Before application design, some research was done to find already existing solutions. The purpose of this task was to find out if we could partially use some other programs and also get a better understanding of how this kind of application could be built.

2.2.1 Wakatime

Wakatime is one of the most popular applications used for tracking your work. It supports a variety of Integrated Development Environments (IDEs) including JetBrains IDEs required by TalTech. It has both its IDE plugins and Command Line Interface (CLI) app licensed under Berkeley Software Distribution 3 Clause Licence (BSD-3). It also has a public API that allows fetching time data from its servers. The downside of using Wakatime is that it costs to see data older than 2 weeks and as it only temporarily stores data on a client machine, there is no easy way to access data without modifying the already existent CLI app. Wakatime also does not make use of Git statistics such as lines added and lines removed. It only reads branch name, commit message and author data. The reason behind this is likely its architecture - it continuously sends heartbeats with required data to its backend. If no internet connection exists, it stores these heartbeats locally in a rather cryptic format. As a result, there is not much data stored locally meaning that the backend has to bundle time data to commits which becomes a very complicated task once users start rewriting history, especially given that Git is a distributed version control system. At the end of 2020 Wakatime also launched its Git integration that allows seeing time spent per commit, but for previously described reasons it gets confused after history rewrites and maps it based on timestamp. This means that for example stashed time is added to a wrong commit. Wakatime's Git integration also does not support statistics such lines added and deleted.

Although Wakatime's front and backend are private there are some open source alternatives. The two most advanced alternatives are Hakatime licensed under Unlicense and Wakapi licenced under GPL-v3. Both of them also have some kind of frontend capable of displaying data similarly to Wakatime's front end main page and Hakatime has even some more graphs. However, neither of them support any groups' system, and they don't have any Git integration.

As both of them have received numerous heavy updates since autumn 2020, they would be very strong candidates to base our system on their already existing functionality, but that wasn't the case when we started with our project.

2.2.2 Git-Time-Metric

Git-Time-Metric is a popular open source time tracking plugin licensed under MIT licence. It has plugins for the most commonly used IDEs including JetBrains IDEs, VSCode, Vim and more. It is closely tied to Git as it stores its time tracking data in Git notes. This allows it to access more Git data such as lines added and lines removed for each commit. Git-Time-Metric does not have a web interface (there are some half-done and abandoned projects meant to run only locally). On the other hand, it has a very powerful and pretty CLI also capable of displaying data in a human-readable format.

One of the biggest downsides of Git-Time-Metric is that its active development has stopped in August 2019 and also its plugins haven't received many updates. Since JetBrains IDEs have changed a lot since that time it doesn't work on the newest IDE versions making it tricky to use.

The biggest design difference between Wakatime and Git-Time-Metric is that Git-Time-Metric stores all its data locally. This makes the Git repository own its time tracking data and the user not to worry about external companies policies. The CLI app is free and as long as there is a free Git server, you don't have to pay for anyone to persist your time data.

2.2.3 DarkyenusTimeTracker

DarkyenusTimeTracker is also an open source time tracker licensed under The Unlicensed. It is only a plugin built for JetBrains IDEs, but it does support automatic recording. It only stores time since the last commit, but it does support automatic time logging to commit messages meaning that this data can be possibly collected by Jira or some other software. Sadly Gitlab, nor Github currently support parsing time data from Git commit messages (Gitlab only supports time logging directly under issue). The other downside is that it only logs the total amount of time spent, no per-file statistics.

Similarly to Git-Time-Metric, it does not have a web interface, and it would be very hard to make one as the time date is not clearly separated from Git commit messages. It also doesn't have a CLI interface due to similar reasons as a web app.

2.2.4 General time tracking applications

There are also several more general time tracking applications, some of which are Toggl, Forest, RescueTime and Clockify. Although they are designed for more general use, it is important to also review their main features to see, if any of them can be extended to be suitable for our use case. [1]

These applications are available for multiple platforms and devices, including phones. Most of them have plugins for different applications, including IDE-s in some cases, but they always require some manual input for starting tracking. Usually, the manual input is required to choose the task and the rest of the tracking is automatic. As these apps are not built for programmers, they lack Git-related statistics such as lines added and removed. The most accurate connection, between time and code written, is usually the task/branch name.

In short, these applications solve the different problem of measuring work time, rather than the problem of measuring code writing statistics.

2.2.5 Existing plugin analysis results

As a result of analysis, a table was drawn to visualize differences between existing solutions. Toggl was chosen to represent general time tracking applications, as it has the most programming-related features out of the ones described above. Features comparison is visible in Table 1.

	Wakatime	Git-Time-Metric	Darkyenus-TimeTracker	Toggl
Supports most common computer operating systems	Yes	Yes	Yes	Yes
Is fully automatic	Yes	Yes	Yes	No
Has Git integration	Only in paid version	Yes	No	Only with issues
Supports tracking non programming activities	Yes	No	No	Yes
Is open source	Partially	Yes	Yes	No
Can be used locally	No	Yes	Yes	No

Table 1. Existing solutions features comparison.

During the analysis there were two main ideas which way to go:

1. Build a Git statistics support for Wakatime.
2. Add some features to Git-Time-Metric and rebuild (some of) its plugins.

Toggl was not viable for us as it is built to solve a different kind of problem. Darkyenus-TimeTracker was ruled out as it lacked features useful for us. Non the less, it was decided that we can use it as a base for building JetBrains plugin.

Wakatime seems to be the best choice for targeting only companies. It is already built for companies to use and it has been tested a lot. It's also more delicate than Git-Time-Metric when handling source code as it supports obfuscating file names and paths.

On the other hand, building a Git statistics support for Wakatime also requires us to build our own backend for all the data or pay to access Wakatime API. Either way, an external database is needed to store all its Git-related statistics. Whilst it has great benefits, it is very hard to implement as it would require adding extra features to all of its plugins. As Git history rewrites may cause conflicts it also needs either a complex system to solve these conflicts or just ignore history overwrites. The second option would mean that whilst the time data saved would very closely reflect developer work in the short term it might get confusing in the long run. As a result of numerous rebases and squashes in different branches it becomes difficult to understand where the time data belongs. The question arises: Why would we want to have time data attached to many commits if we don't really care about the commits and just squash them? For us, the answer was that we should also squash the time data to only "point" to a single squashed commit, but it is too complex to achieve.

When adding Git statistics to Wakatime it is rather hard to persist its security features such as obfuscating file names. As a result, whilst Wakatime is currently the most ready for commercial use, it is hard to implement extra features.

Adding features such as automatic notes fetching and pushing to Git-Time-Metric on the other hand is way simpler. Whilst the rewriting of IDE plugins is a lot of work it's still less than building a system for Wakatime capable of handling Git rebases. It is also important to mention thant Git-Time-Metric architecture makes it difficult to record indirectly programming related tasks time such as searching for help online. The reason for it is that it is very inconsistent to tie time spent outside editing Git repository files to certain Git repository. In this project context, we don't see this as a big problem as we are mainly targeting time spent inside IDE. Trying to record internet browsing data or time spent on other programs may be seen as an invasion of privacy, and it might greatly reduce the number of users interested in our app.

3. Project design

We decided to base our application on Git-Time-Metric and have the time tracking data stored in Git Notes for every repo so that it can be tightly tied to code. This gives us an opportunity to have a more complete overview of when and on what issue/branch the time was spent. It also gives access to features such as Git diff that can be used for measuring writing speed. Furthermore, it lifts a job from our shoulders by handling complex cases such as rebase and merge for us.

The downside is that Git notes take up some space on the client machine(s) and Git server. They can also be externally edited, and they might cause issues with Git usage if the developer does not follow best practices. Nonetheless, we decided to go this way as the space used by Git notes is fractional compared to file sizes, and our app is a tool for the developer, so we don't see any urgent need to protect it from him.

Later it was brought out in Gitlab time tracking issue discussion, that they consider storing time data in Git notes to be a very clean approach compared to storing it in commit messages. However, they have not yet decided to implement it themselves as it requires major changes in their current business logic. [2]

Partly enforced by the usage of Git, the general design is shown in Figure 1.

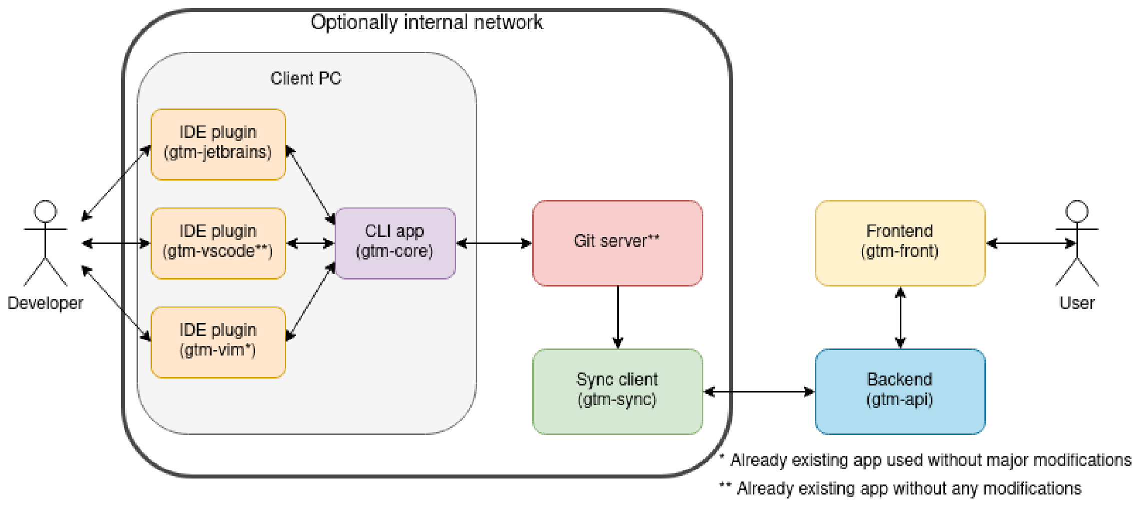

Optionally internal network
Client PC
IDE plugin (gtm-jetbrains)
IDE plugin (gtm-vscode**)
IDE plugin (gtm-vim*)
CLI app (gtm-core)
Git server**
Sync client (gtm-sync)
Frontend (gtm-front)
Backend (gtm-api)
Developer
User
* Already existing app used without major modifications
** Already existing app without any modifications

Figure 1. Application general design

The general time-tracking application can be divided into three different kinds of smaller applications:

1. **Applications that are responsible for data collecting.** These applications are CLI app and IDE plugins and are installed on the developers machine.
2. **Applications that are responsible for syncing data.** This role is filled by a Sync client. These applications can be installed onto companies/universities servers where they can access Git's internal network if needed.
3. **Applications that are responsible for analysing and displaying data.** These are Frontend and Backend apps that are installed on any server.

The reasoning behind this architecture is security and scalability. For every backend there can be multiple Sync clients all installed on different machines and with different Git access rights. For every Sync client, there can be multiple client apps all logging time independent of one another. And in the end, there can be multiple IDE plugins interacting with CLI app.

For security, it is very important to ensure, that no sensitive information is leaked. To deal with this issue many companies have their Git available only in an internal network. Not to raise any more security concerns we decided to follow the same principle and not export any Git code outside an internal network. That is the reason, why Sync client is needed as it has access to the internal network, but only uploads time-tracking related information to our backend. Therefore, actual code never leaves the internal network.

All the applications that are installed on either client machine or client's server are open source so that the client can easily verify, they only do what they are meant to do and perform no malicious activities.

To properly manage numerous apps in different programming languages we decided to have each app source code in a separate Git repository. This reduces the number of merge conflicts and also makes building CI/CD easier.

3.1 Client CLI app

This is the main app on the client machine. The app is responsible for storing events sent by an editor to files and then later also combining the stored information into Git Note. You can also see all main stats, such as time spent on a commit via CLI app.

The app is based on open source time tracking app Git-Time-Metric written in Go and licensed under MIT licence. We decided to base our app on Git-Time-Metric solution because their app was working the way we wanted to, it had plenty of users meaning plenty of testing done, and the Git-Time-Metric app had a good code style.

Gtm-core folder structure is displayed in Table 2.

Folder	**Purpose**
.github	Github related files (Workflows for CI/CD, issues and pull requests templates)
deploy	Deploying related files. Currently, only Windows installer files and Licence packed with installer.
command	Every command has files *<command>.go* and *<command>_test.go* which respectively are controller for command, and it's tests. The controller is responsible for parsing arguments, calling appropriate services and printing out results.
docs	Documentation files.
epoch	Unix epoch helper functions and their tests.
event	Events files serialization and deserialization related services and tests.
metric	Metric file serialization, deserialization services and tests. Metric files are used for storing time data of files added to Git, but not committed.
note	Git notes serialization, deserialization and tests.
project	Gtm initialization and uninitialization related services with tests. These services are responsible for calling appropriate functions from *scm* package to Git hooks and modifying Git config.
report	Report generation services used by report command.
scm	Git related services. These services are used to wrap *libgit2-go* services into more usable form.
test	Python files for running stress tests for gtm-core.
util	Various string, date, time and math helper functions.
vendor	Directory containing dependencies listed in Git submodules.
<root>	Program entry point, Git related files, README and Gofmt style lint configuration .

Table 2. Gtm-core folder structure.

Although the app was already working we still needed to do many bug fixes and add some features it didn't have. The biggest change in design was that we moved copied in dependencies to Git submodules. Although this seems like a small step, it turned out to be very complex to build Go app linked with C library dynamically. As we were not able to get all Git submodules dependencies to work properly in Windows environments, we decided to leave it separately fetched and built in CI/CD workflow file. Nonetheless, all "snapshot" dependencies were removed. We also added stress tests to verify the effect of using gtm-core with large amounts of commits.

3.2 IDE plugins

IDE plugins are installed on developer IDE, and they execute CLI app commands on specific editor events. Plugins are needed to listen for editor events such as typing without having to give CLI app extensive permissions to run in background. They also provide a simple way to display some information in IDE. For example, time since the last commit is shown to the user inside IDE.

Currently, we only have one plugin that is compatible with all Jetbrains IDE's. The gtm-jetbrains plugin is written in Kotlin and uploaded to Jetbrains plugin repository so that you can easily install it in your Jetbrains IDE. Kotlin was chosen as the plugin was limited to Java Virtual Machine (JVM) based language and only Java and Kotlin were widely supported. We have experience in both Java and Kotlin, but we both preferred Kotlin to Java due to its null safety and functional patterns.

Gtm-jetbrains folder structure is described in Table 3. Namespace *ee.developest.gtm* is shortened to <*ns*> for readability.

Folder	**Purpose**
.gradle	Gradle related files
src/<ns>/listener	Editor event listeners.
src/<ns>/popup	Pop-ups related controllers. Used for getting user input
src/<ns>/service	Service files such as ConfigService.kt
src/<ns>/widget	Widget factories used to display time since last commit and some feedback about initialization.
src/<ns>	Gtm-core wrapper used to forward commands from listeners to gtm-core and gtm-core to popup / widget.
<root>	Gradle configuration, README and LICENCE.

Table 3. Gtm-jetbrains folder structure.

3.3 Sync client

Sync client is run on a network, where it can access Git repositories. On Git push, Git webhook sends a request to gtm-sync via HTTP request. Then gtm-sync fetches Git repository with its time data, extracts required data and syncs it up to Backend.

The gtm-sync application was written in Rust as it had a more up-to-date library for *libgit2* than both Java and Go, required for interacting with Git. Although we did not have prior experience with Rust we preferred it to C/C++ as it is memory safe and has higher-level libraries that can be used for building a web server and API client. Python and NodeJS that also provide convenient higher-level functions were ruled out because they produce very big memory footprint compared to Rust, and they also run a lot slower. Neither of them also gives a type safety that was a must for us.

For the code design, we followed domain-based architecture. Folder structure for gtm-sync is displayed in Table 4.

Folder	**Purpose**
.github	Github workflows.
src/config	Config serialization, deserialization and config helper functions.
src/gtm	GTM notes parsing and Git related services.
src/repo	Tracked repository managing services.
src/server	Rocket controllers used for IO.
src/sync	Syncing data with gtm-api related services.
<root>	Git related files, Cargo package manager files, README and LICENCE.

Table 4. Gtm-sync folder structure.

3.4 Backend

Backend is a collection of code that runs on the server. It receives requests from clients and sends appropriate data back to the clients based on the business logic. Backend also has a database that stores all the necessary data for the application.

In our case the backend receives data from the sync client via REST API requests and stores the data in the database. End users can interact with the backend though frontend via REST API requests.

The backend application has been built with Rust programming language using Rocket framework. Rust is a relatively new low-level statically-typed programming language that is focused on performance and safety. It gives almost the same speed as C, but guarantees both memory and thread-safety. It also follows many more functional patterns than Java/C/C++/GO, which makes writing code easier and safer once mastered. Since we needed a fast backend with minimal overhead for processing large datasets, Rust was the best choice. Java, C/C++ and GO were also considered, but the lightweightness of Rust overweights Java, whilst memory safety overweights C/C++. Since Rust was chosen for sync client, we decided to choose it for the backend as well.

3.4.1 Backend design

Backend code base has been built following Domain Driven Design (DDD). DDD is a concept that centers the development and code structure around the business domain. It helps to keep the code more organized, maintainable and extendable [3].

A layered architecture can be found on Figure 2. Controllers are used for handling HTTP requests and parsing parameters. Once the parameters are parsed, appropriate method is called from service layer. The service layer is responsible for business logic. To store the data into the database, appropriate repository's methods are called as all database operations are done in the repository layer. Some service methods also do external API requests. When the controller receives a response from the service, it encodes it and sends it back to the client.

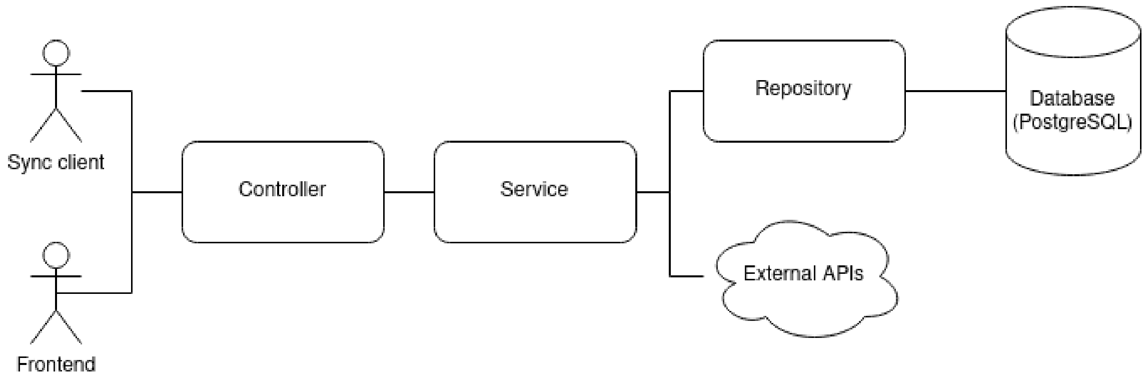


Figure 2. Backend architecture

Folder structure is displayed in Table 5.

Folder	**Purpose**
.github	CI/CD related files
migrations	Migrations for Diesel.
src/common	Widely used files.
src/db	General database connection.
src/domain	Main business logic.
src/security	Authentication and authorization.
src/vcs	Version control.
<root>	Configurations, README and LICENCE.

Table 5. Gtm-api folder structure.

3.4.2 Database design

One requirement for choosing the database, was that we needed it to have bindings for our Object-relational mapper (ORM), so we could easily use it in our application. This requirement narrowed the search down to SQLite, MySQL and PostgreSQL as they were the only languages with a full support in Diesel. There are other ORM-s written for Rust, but Diesel is one of the most popular ones, so we decided to go with it as there was no need to use some other database.

SQLite was ruled out as it is not strongly typed, and does not have the same amount of features PostgreSQL and MySQL have. For deciding between MySQL and PostgreSQL it came down to personal preference as both of them are strongly typed and support all the required features, including recursive queries. Both of them are also open source.

We decided to go with PostgreSQL, as we had prior experience with it.

As we are running the application in "beta", we decided to run the database in a docker container on our development machine to reduce the cost of upkeep. For data persistence, we have mounted database files volume to the host machine.

Database versioning

As we have users data in our database, it is important to also persist it through database schema updates. For this, we use migrations and let Diesel handle the updates. Every time database change is required, we add a new migration with both up and down scripts that can be used by Diesel to run the migration or revert it. Diesel is configured to automatically run updates added to the release. As reverting database changes can result in a loss of data, it has to be done manually via Diesel CLI app.

ERD Schema

The database holds data about tracked time, registered users and how the users can access time data. The structure of the database schema is described in Figures 3 and 4.

Time data follows the structure of the Git commits tree and file tree. The *repositories* table contains general information about the Git repository such as name and the sync client responsible for tracking it. The *commits* table has the information about commits such as author, timestamp and commit message. Each *commits* table entry also has a reference to the repository it is committed to. Every commit has zero or more files that were edited in it. These files are contained in *files* table, and they contain information about files added and deleted. To have more accurate data about when the time was spent, separate table is needed as there is no guarantee that the time spent editing files is spent just before the commit timestamp. For this, we have a table called *timeline*, which holds timestamps and durations of every consecutive edit as well as a reference to a file in which the time was spent.

The *users* and *roles* tables are responsible for persisting user login credentials, roles and connected OAuth accounts. As we allow the user to optionally register only via OAuth, the password field is nullable. Users that have no password, can only log in via OAuth, but they can also add a password at any time.

Every user can have multiple rows with role *USER* being added to every user by default. There are three roles in total: *USER*, *LECTURER* and *ADMIN*. Currently, there are no rules in the database for any of the roles, which means all checks are done in software. More detailed description about roles can be found in Roles subsection.

Every user also has zero or more OAuth logins stored in the *logins* table. There is a limit that every user can have at maximum one OAuth login per login type. If one already exists, it is updated instead of adding a new.

User emails retrieved from OAuth providers are stored in the *emails* table to avoid unnecessary web requests.

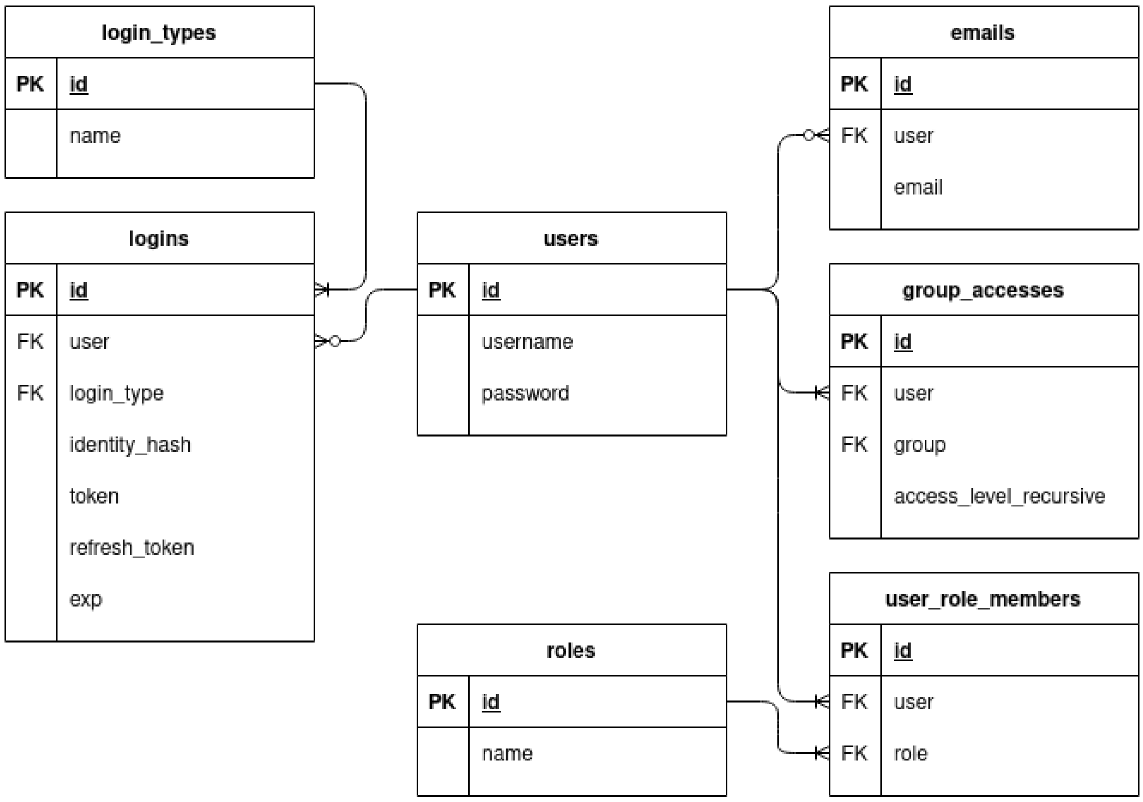


Figure 3. ERD schema for user related tables

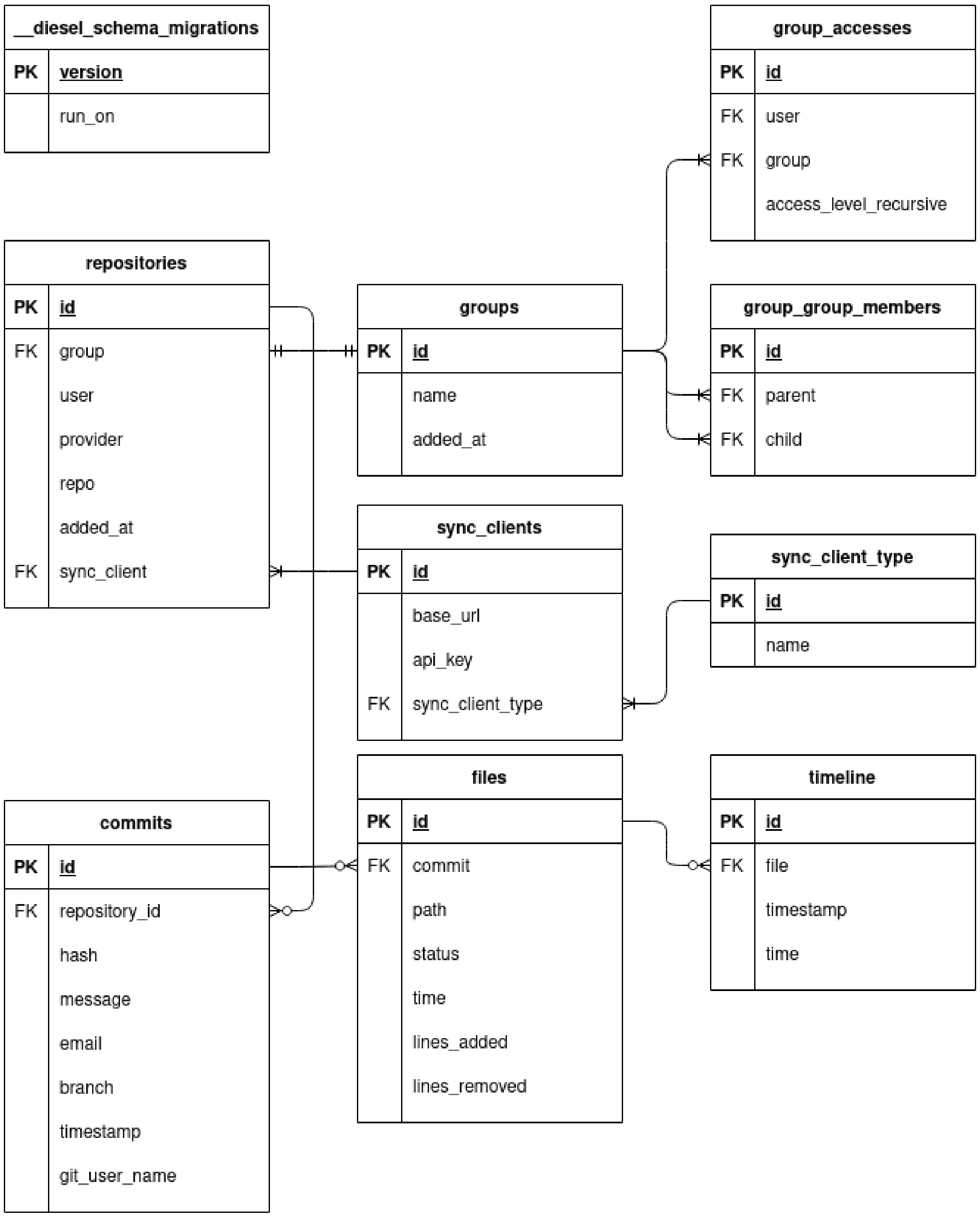


Figure 4. ERD schema for non-user related tables

3.5 Frontend

A frontend is a software program or website that the user interacts with. In our case, frontend is a website where users can see time tracking data according to their privileges. Although technically also CLI app can be considered frontend, when we refer to frontend we only mean the website.

The frontend has been built using ReactJS library. According to the State of Frontend 2020 report, ReactJS is the most used JavaScript framework and also the framework, which developers want to learn or keep using the most. It is faster than Angular, and its somewhat functional style was more appealing for us than the style of Vue. It was chosen by Marten, who is mostly responsible for frontend development and has had some prior experience with it. The most popular frontend frameworks are shown in Figure 5 sorted from most favored to less.

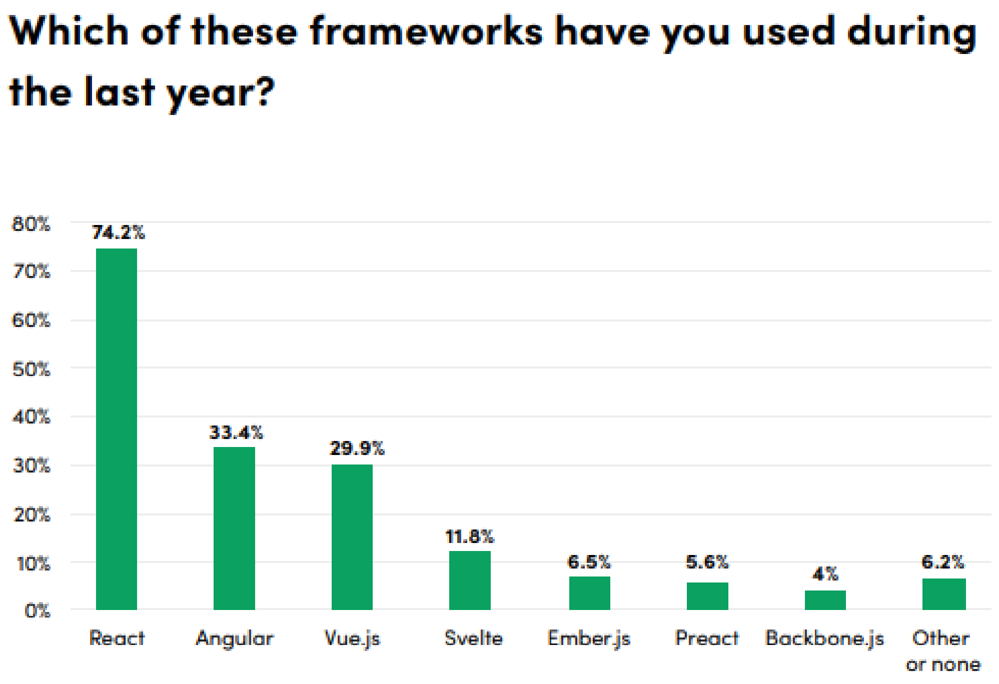


Figure 5. Frontend framework popularity [4]

3.5.1 Code design

Frontend code is written following the main principles promoted by ReactJS community and documentation. For an example, component's names use *PascalCase*, helper files use *camelCase*, folders use *camelCase*. Code is divided into multiple files/components to prevent code repetition, inline CSS is avoided, the liner is used to make code easier to use, service calls are in the separate file, etc. To improve code quality and readability TypeScript was chosen, as it is a strongly typed language, which JavaScript is not. This decision was not made at the beginning of the project but as the code kept growing and readability suffered the decision was made. One of the biggest reason for the switch from JavaScript to TypeScript was that due to being a strongly typed language, TypeScript has better IDE support. Folder structure is displayed in Table 6.

Folder	Purpose
.github	CI/CD related files
src/api	Service calls .
src/assets	Images and icons.
src/containers	Main container structure (Header, Footer, Content, etc.)
src/models	Easy-Peasy state management models.
src/reusable	components that can be reused.
src/routes	Routes.
src/scss	Style.
src/store	Store for state management.
src/utils	Helpers.
src/views	Views that are used once.
<root>	TypeScript configuration, package.json, README and LICENCE.

Table 6. Gtm-front folder structure.

3.5.2 User interface design

User interface must be built keeping the user in mind. User experience is the most important part of a web application. If the user does not have a good experience with the interface, it might be the end of the interaction between the user and the application. Since the team did not have any professional designers, CoreUI was a great choice to make the design process much easier. CoreUI is an Open Source Bootstrap Admin Template which is built on Bootstrap and written with readability in mind. The design and easy usability were the decisive factors in choosing CoreUI.

However, CoreUI's charts were not customizable to the extent we required. For better customization, Recharts was chosen for building graphics. Recharts is a composable charting library built on React components and D3. It is much more customizable than CoreUI charts as it allows combining different bar charts, line charts, etc. [5]

At this point, the design problem arose again. Designing graphs were done in the following manner:

1. Understand the functionality and create an initial component.
2. Let it settle for some time and then discuss with the team.
3. Finalize component and release it for test users.
4. Make final changes according to the users' feedback.

Design patterns such as graph sizes, colours, spacings, fonts etc. were chosen in such a way that they would look similar to CoreUI graphs.

3.5.3 State management

State management does not have a certain winner. Choosing a state management library comes down to personal preference. We decided to use Easy-Peasy, which is built on the most popular state management library Redux. Easy-Peasy was chosen over Redux because Redux has a steep learning curve but Easy-Peasy is easy to use.

4. Project contents

4.1 CLI app

4.1.1 CLI app improvements

Our CLI app is based on Git-Time-Metric, an already working open source time tracking app. Although it already had all the basic features, some improvements and fixes were required.

Firstly *–cwd={some/path}* option was added to allow easier integration of plugins. Option to pass a current directory as an option is required, as IDEs not launched from terminal sometimes have their working directory elsewhere and setting custom working directory every time we call CLI app is more complicated than passing an argument.

To clean up user home directories, global configuration files were moved from */.git-time-metrics/* to */.config/gtm/*. For Windows, a new installer was built to make installing easier. Also, it was chosen to switch from statically compiling SSH2 Libraries into our application to dynamic linking. For Debian Linux, a build script for Debian packages was set up to also provide Debian packages with every release.

The application was updated to automatically add a fetch refspecs to fetch data from remotes on Git fetch. Also, a pre-push Git hook was added to automatically push data to origin. With these two hooks, the time data is automatically fetched and pushed whenever Git push is made requiring the user no extra effort.

To still allow users to only track time locally, a *–local* option was added to *init* command. Also, *–auto-log=[jira|gitlab]* option was added to support automatic time logging to commit messages. Currently, this time can only be collected in Jira, but there is also and issue open for Gitlab to implement this.[2]

To allow filtering data by branch name we decided to also store the current branch name in Git Notes when committing a file. This could not be avoided because the branch is simply a pointer to commit and there is no guarantee that this pointer is not changed or deleted.

For students, it is important to view time spent on specific tasks. In most subjects, the tasks are placed in separate subdirectories for better file management. To allow users to view time spent on specific tasks, a *–subdir="sub/dir/name"* option was added to filter out data based on directories.

Although Git supports automatic merging/rewriting of the notes, it was inconsistent in some cases, especially when the user wants to also rewrite manually added notes. To keep time data after Git rebases, a new command *rewrite* was introduced. This command is run by Git post-rewrite hook, and it rewrites notes so that they still point to the correct commit.

4.1.2 Initializing tracking

Main flow for initializing tracking for repository is described in Figure 6.

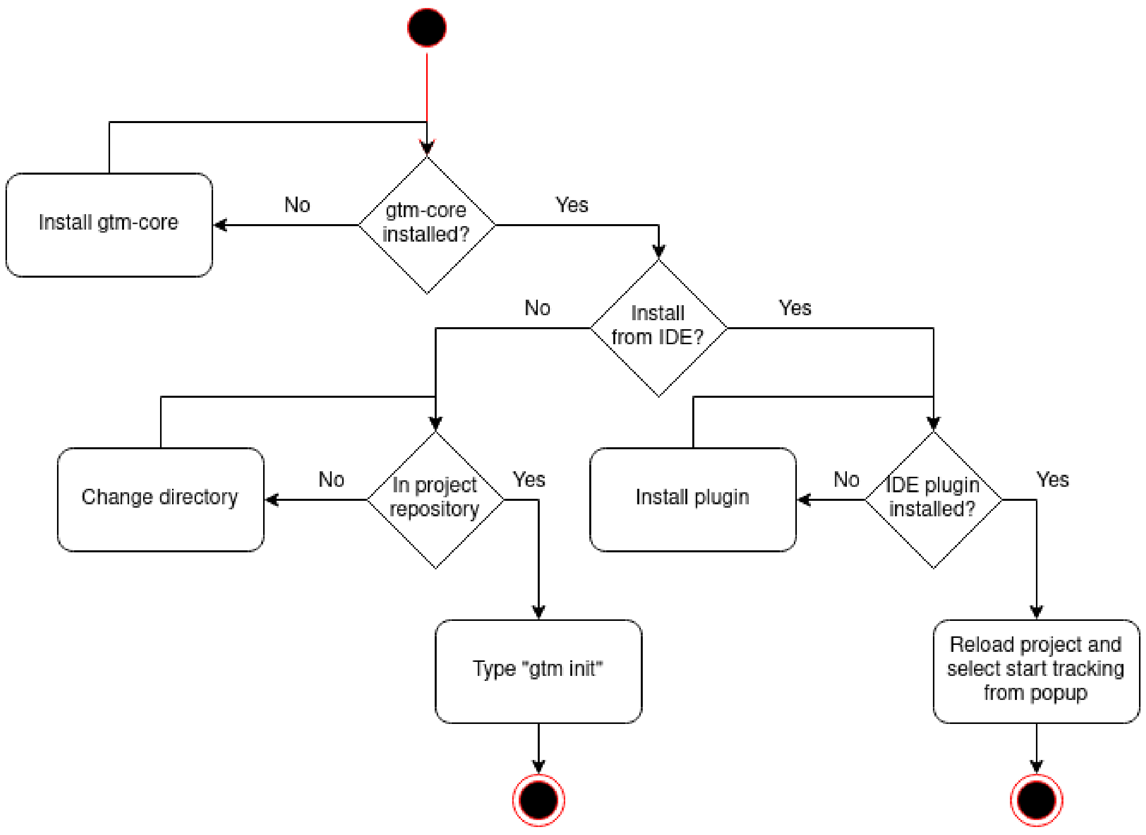


Figure 6. Initialize tracking

The first requirement is that you need to have gtm-core installed and added to *PATH* in order to initialize tracking. If the user has gtm-core installed, he can initialize tracking from command line. To do that, it is required to change your current directory to be somewhere within Git root directory. Time tracking is then initialized with *gtm init* command. The command supports following arguments:

Argument	Description
–terminal	Enable time tracking for terminal. (Requires terminal plugin)
–auto-log	Either *gitlab* or *jira* to automatically log time into commit messages with given format.
–local	If set, no Git fetch nor push hooks are added. Time data is only stored locally.
–tags	Optionally add tags to project for better local organization.
–clear-tags	If project already has some tags running it with given flag removes previously added flags.
–cwd	Useful for specifying working directory for scripts.

Table 7. Gtm-front folder structure.

It is safe to run *gtm init* multiple times with any of the arguments.

The user can also initialize time tracking from within IDE if he has GTM plugin installed. Currently, only Jetbrains IDE plugin supports starting tracking. To initialize time tracking from Jetbrains IDE the user needs to open/reload the project he wishes to add time tracking to. Upon doing that he is prompted to choose whether to start tracking or not. To initialize tracking, start tracking has to be chosen. This initializes tracking for the currently open repository with default settings (equal to no argument for CLI). If the user chooses not to initialize tracking, his choice is persisted in a local Jetbrains project config file within *.idea* directory. To later add tracking to a repository, he needs to do it from the command line or by deleting the Jetbrains' project settings file.

4.2 IDE Plugins

IDE plugins are used to capture editor-specific events and then execute appropriate commands on gtm-core. Essentially gtm-core and IDE plugin combo works like a language server and code formatting plugin combination. This eliminates the need to write duplicate code, when writing plugins for multiple IDEs. Currently, we have full support for Jetbrains and some support for Vim. There are also many more plugins that have been previously written for original Git-Time-Metric, and most of them also work with our app.

4.2.1 Jetbrains

Jetbrains also had a plugin for the original Git-Time-Metric, but the IDE has gone through lots of changes since the last update to the plugin. As of 2021, this causes Jetbrains IDEs to crash. We decided to fully rewrite the plugin as it used a lot of deprecated Jetbrains components.

The plugin catches following events and forwards the data to gtm-core:

1. Opening the file in the editor.
2. A mouse pressed inside the editor (on file).
3. A process run (Mostly build, test, debug or run, but can be any other process you run via IDE)
4. File saved.
5. Visible area changed (Window resized / split).

In response, the plugin receives the time since the last commit (uncommitted time), which it shows on the bottom status bar.

In addition to silently listening for events and updating time, the plugin communicates with the user via pop-up dialog and notification. The pop-up dialog is used to prompt the user whether he wants to start tracking time for the project as described in Initializing tracking Notifications are used to report errors and more important action results (such as whether adding tracking was successful).

4.2.2 Vim

An already existing plugin for Vim was forked as it had all the required features. Version numbers and update links were changed to comply with our version. As the Vim script file to give the support for tracking time and also displaying it on the status line is under 80 lines long it perfectly describes, why it was necessary to have gtm-core as a separate program instead of writing the business logic inside the plugin.

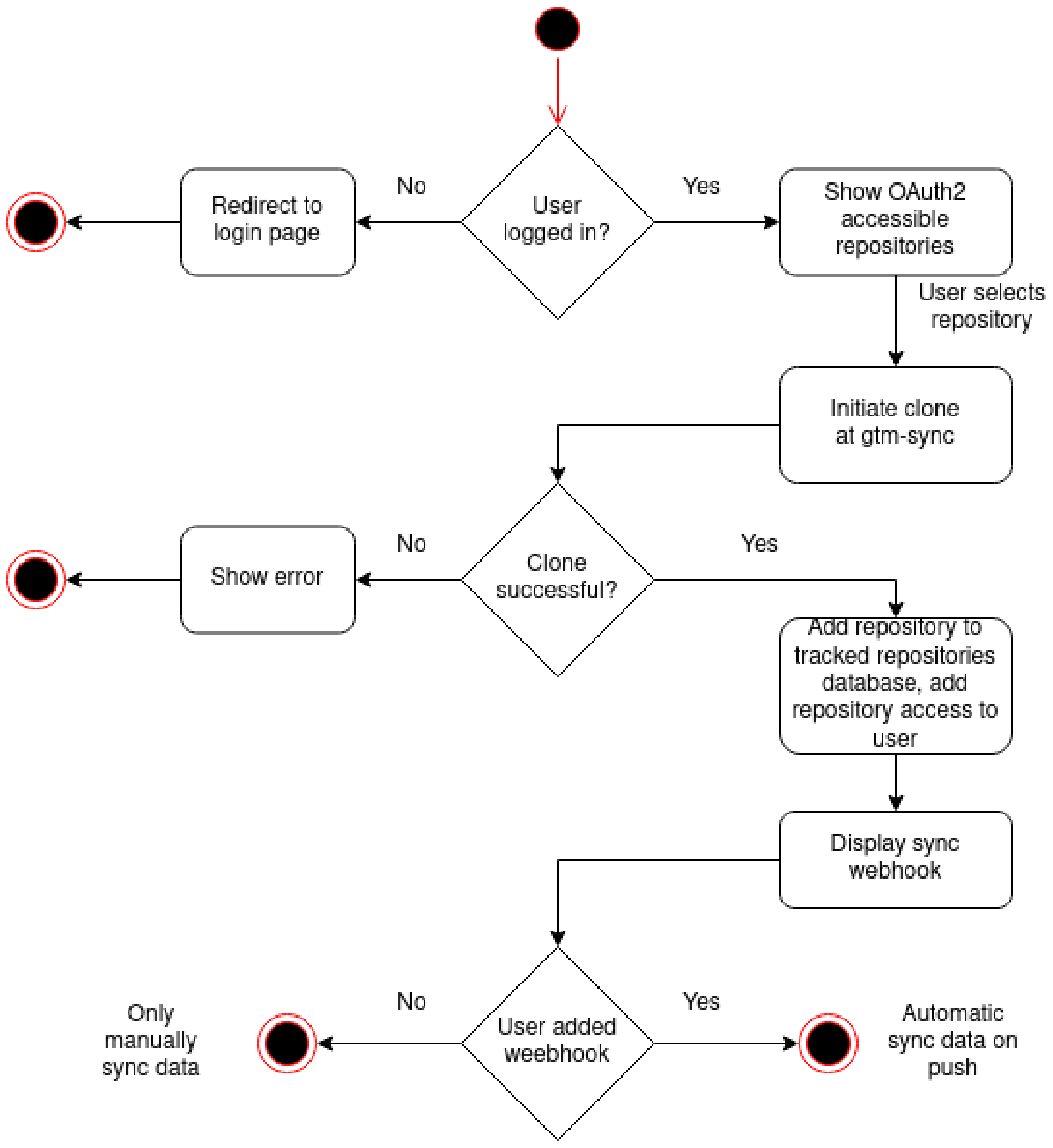
4.2.3 Adding tracking to repositories

We have added three levels to view time data:

1. Add tracking (locally), view data via GTM CLI app.
2. Add tracking (locally), log data to commit messages.
3. Add tracking, sync data to GTM Web App.

All smaller level features can be included while using a higher level option. Viewing data via CLI app is always possible, as it is the same app that is responsible for recording data. The data is stored in Git notes, and it does not matter whether the notes are local or not for the CLI app to read them. You can also add logging time spent to commit messages in both of the scenarios as it simply works on top of the already existing system and only modifies the commit message. To use auto logging option you simply need to add *–auto-log=[gitlab|jira]* option when initiating tracking.

To view data from the web interface, further steps are required. User flow for making repository visible on GTM web application is shown in Figure 7.



User logged in?
No
Yes
Redirect to login page
Show OAuth2 accessible repositories
User selects repository
Initiate clone at gtm-sync
Clone successful?
No
Yes
Show error
Add repository to tracked repositories database, add repository access to user
Display sync webhook
User added weebhook
No
Yes
Only manually sync data
Automatic sync data on push

Figure 7. Add tracking user flow

Firstly, to have the sync client access the data, time data cannot be only stored locally. To add a repository, the user needs to have linked appropriate Git server provider account via OAuth so that we can verify that you have the access to the repository. To add a repository, the user needs a linked OAuth account from the appropriate Git server provider. The OAuth login is used to verify the user's access to the repository. After that, the user need's to search for the repository from the web interface and click *"Start tracking"* button. With that, the tracking is technically added, but to also automatically sync time data on every push to remote, the user needs to add a webhook to the Git repository.

To make sure that the repository can be accessed by the user the request to an appropriate provider is made again when the user has pressed the *"Start tracking"* button. This check is required as the user could easily guess other peoples private repository clone URLs and then post it to our backend with his own JWT. This extra check also means the user cannot add tracking to a self-hosted Git server that does not provide OAuth2. However, we were not able to find any better way to identify that the given user has access to some Git repository and therefore it was decided to not support any custom setups.

4.3 Sync client

Sync client is used for fetching data from Git notes and then syncing it up to the backend. The gtm-sync application is configured in a local configuration file, and it exposes some REST API endpoints that allow adding new repositories and endpoints for triggering syncing process

Manually editing configuration is meant for a developer as it gives more control over application and REST API is for machine to machine communication. All the API endpoints are described in Table 8.

Endpoint	**Description**
GET /repositories/<provider>/<user>/<repo>	Used to get data about single repository
POST /repositories	Used to start tracking repository
GET /repositories/sync-all	Used to sync all tracked repositories to backend
GET/POST /repositories/<provider>/<user> /<repo>/sync	Sync single repository. Post is allowed as webhooks sometimes only support post requests
POST /hooks/github/push	Endpoint for Github hook to sync repository
POST /hooks/gitlab/push	Endpoint for Gitlab hook to sync repository

Table 8. Gtm-sync API endpoints.

For configuration *config.toml* file is used. It contains information about the backend, where to sync data, Git user authorization (SSH keys), and about tracked repositories. Although gtm-sync supports cloning with both Secure Shell Protocol (SSH) and HTTPS, we have limited it to HTTPS in our backend for simplicity. In order for SSH keys to work, we have created a user called *gtm-user* to TalTech Gitlab server and added an SSH key to him. This means *gtm-user* has to have at minimal reporter access to the project in order to track it. For GitHub and GitLab we only support tracking public repositories for now and for the SSH cloning to work, the SSH key was added to Tavo Annus'es personal account. The configuration variables are described in Table 9.

Variable	**Description**
target_base_url	Backend URL, used for syncing data
sync_base_url	Public URL to self (Optional), used to force sync
access_token	Backend API key
ssh_public_key	Path to Git user public ssh key
ssh_private_key	Path to Git user private ssh key
repositories_base_path	Path to store cloned repositories at
repositories	List of tracked repositories (each has path in filesystem and clone URL)

Table 9. Gtm-sync config variables.

4.4 Backend

Backend is the central part of our system as it connects frontend, sync client, and database. It communicates with gtm-sync and gtm-front via REST API requests.

Sync client which has API key registered has access to the endpoints in Table 10.

Endpoint	**Parameters**	**Description**
POST /repositories	-	Sync client can send new repo timeline data
PUT /repositories	-	Sync client can send existing repo updated timeline data
GET /commits/hash	provider: string, user: string, repo: string	Returns last commit hash, timestamp and tracked commit hashes

Table 10. Gtm-api endpoints for sync client.

An authorized user has access to the endpoints in Table 11.

Endpoint	Parameters	Description
GET /groups	-	Returns all the groups logged-in user has access to.
GET /{group_name}/activity	start: int, end: int, interval: string, timezone: string	Returns activity for group.
GET /{group_name}/subdirs-timeline	start: int, end: int, interval: string, timezone: string, depth: int	Returns sub-directory timeline for group.
GET /{group_name}/timeline	start: int, end: int, interval: string, timezone: string	Returns timeline for group.
GET /groups/{group_name}/stats	start: int, end: int, depth: int	Used to get group stats for leaderboard view.

Table 11. Gtm-api secured public endpoints

Unauthorized user has access to the endpoints in Table 12.

Endpoint	Parameters	Description
POST /auth/login	-	For login
POST /auth/register	-	For registering new account

Table 12. Gtm-api not secured public endpoints.

All the endpoints can be found here.

4.4.1 Security

Our application holds data, that shall not be visible to all clients and therefore some kind of authentication and authorization methods are required. For the data stored in Git notes, we decided no extra security is required, as the time data is not more sensitive than the actual code. The security of source code stored in a Git repository is handled by a clients themselves and Git providers. If they wish to have additional protection for Git notes, they can configure it themselves. We only provide the option to have notes only stored locally (not pushed to origin).

For the web application, we needed to implement our own security measures. For the most basic usage, we have username and password authentication. Accounts that only have password authentication are not authorized to access any groups nor repositories unless the Admin user explicitly gives them access to any.

To get automatic access to the repositories the user is a contributor of, the user needs to authenticate himself via OAuth2 standard. This way the application can verify, that the user signing in to the application is the same user that has access to the Git repository.

OAuth2

For OAuth2 authorization rocket_oauth2 library is used, that follows RFC-6749 standard [6, 7]. We support authentication via Github.com, Gitlab.com, Bitbucket.org, Azure (Microsoft account) and TalTech gitlab server. The first three were chosen as they are the most common Git server providers as of 2021. Authentication via Microsoft was added as TalTech (and also numerous other universities/companies) use it and therefore all students have already registered accounts there. Also, it provides access to user emails, that can be later used to filter out user commits. TalTech Gitlab server was added, as the application is currently developed for TalTech and therefore it was a requirement that everything shall also work on TalTech Gitlab server.

From all OAuth2 providers at least user read access is required to get access to user emails, which we use to link users with commits. For OAuth2 providers, which are also a Git server providers also permissions to read user repositories data are required. User repositories data is used to give user automatically access to his own repositories and also display repositories not currently tracked by GTM. It also allows adding automatic data collection to user repositories more easily, as a user can browse through repositories from different Git server providers.

4.4.2 Groups system

One of the requirements for our app was that it should be possible to group repositories together. The grouping is needed to view statistics for multiple repositories at once and also compare groups of repositories. The grouping system should be also capable of controlling user access rights to different repositories.

Requirements for the grouping system were:

- It shall be possible to group both repositories and groups already consisting of repositories into a bigger group.
- Single repository may belong to multiple groups.
- Group access can be limited to only viewing summary (group total).
- Granting user an access group's subgroups and then removing it shall have no side effects. (Accesses to subgroups shall not be persisted)

As the groups may consist of both groups and repositories we decided to automatically create a group for every repository. This means new groups can only consist of zero or more other groups. If we need to get the repository, we can fetch all the group ids that are accessible and then query from repositories database by comparing repositories group id against previously fetched ids. To fetch a group with all of its subgroups a recursive Structured Query Language (SQL) query was needed as the groups' hierarchy formulates a tree-like structure. Our database provider PostgreSQL supports recursive queries so there were no technical problems with implementing it on SQL database. A simplified version of our group system database tables can be seen in Figure 8.

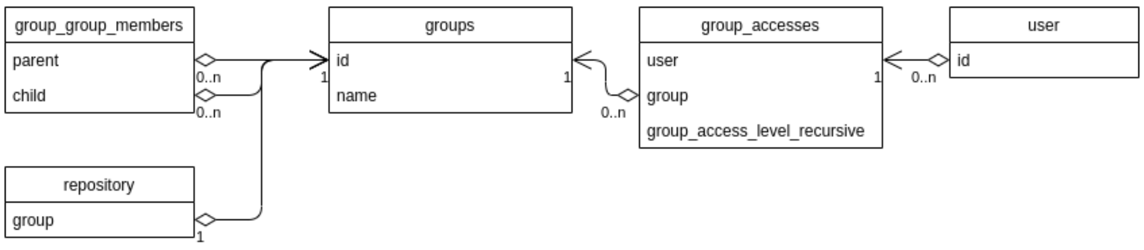


Figure 8. Application groups system

This tree-like groups hierarchy allows to easily give and take user access to any group. If the need to change access from only parent group to also all subgroups access, it only needs to be toggled at the access_level_recursive variable. If one particular group access is removed, all other accesses remain in place, meaning that groups previously accessible via some other group access remain accessible via the other group access.

4.4.3 Roles

Our application has three different roles to control user permissions. A single user can belong to multiple roles and also new roles can be dynamically added to the database as they are kept in a separate table. Currently, there are three roles in total: *USER*, *LECTURER* and *ADMIN*. Role *USER* is added to every created user, and it has no extra permissions.

People with *USER* role can:

1. View data from Git repositories, they have access to.
2. Add new repositories.
3. Delete repositories they have access to.

People with *LECTURER* role can:

1. View data from Git repositories, they have access to.

2. Add new repositories.
3. Delete repositories they have access to.
4. Give access to others.

People with *ADMIN* role can:

1. Assign new roles to other users.
2. View data from all the repositories and groups.
3. Give other users group accesses.

4.5 Frontend

Frontend is a one-page web application. Users can log in and see their data according to their privileges. This application has two main parts - profile and data visualization. Under profile users can change password, link OAuth accounts, delete account and add repositories. Data visualization is divided into 3 parts - *"Dashboard"*, *"Leaderboard"* and *"Timeline comparison"*. In those tabs users can see their/others data as graphs and tables.

4.5.1 Authentication and authorization

Authentication and authorization is implemented using a bearer token system. The user gets JSON Web Token (JWT) from the API and stores it in the local storage and adds it to every request header. The user must log in from *"Log in"* page. Login page view can be found in Figure 11. On success, client receives JWT from the backend and stores it in local storage. The user is then automatically redirected to the main page. Users can log in with OAuth or username/password. In the case of OAuth, the user is redirected to the OAuth platform for authentication. If the authentication is successful, the user is redirected to the frontend with JWT. JWT is saved on local storage, and the user is redirected to the main page, where he can access their data. Login and register user flows are described on Figure 9.

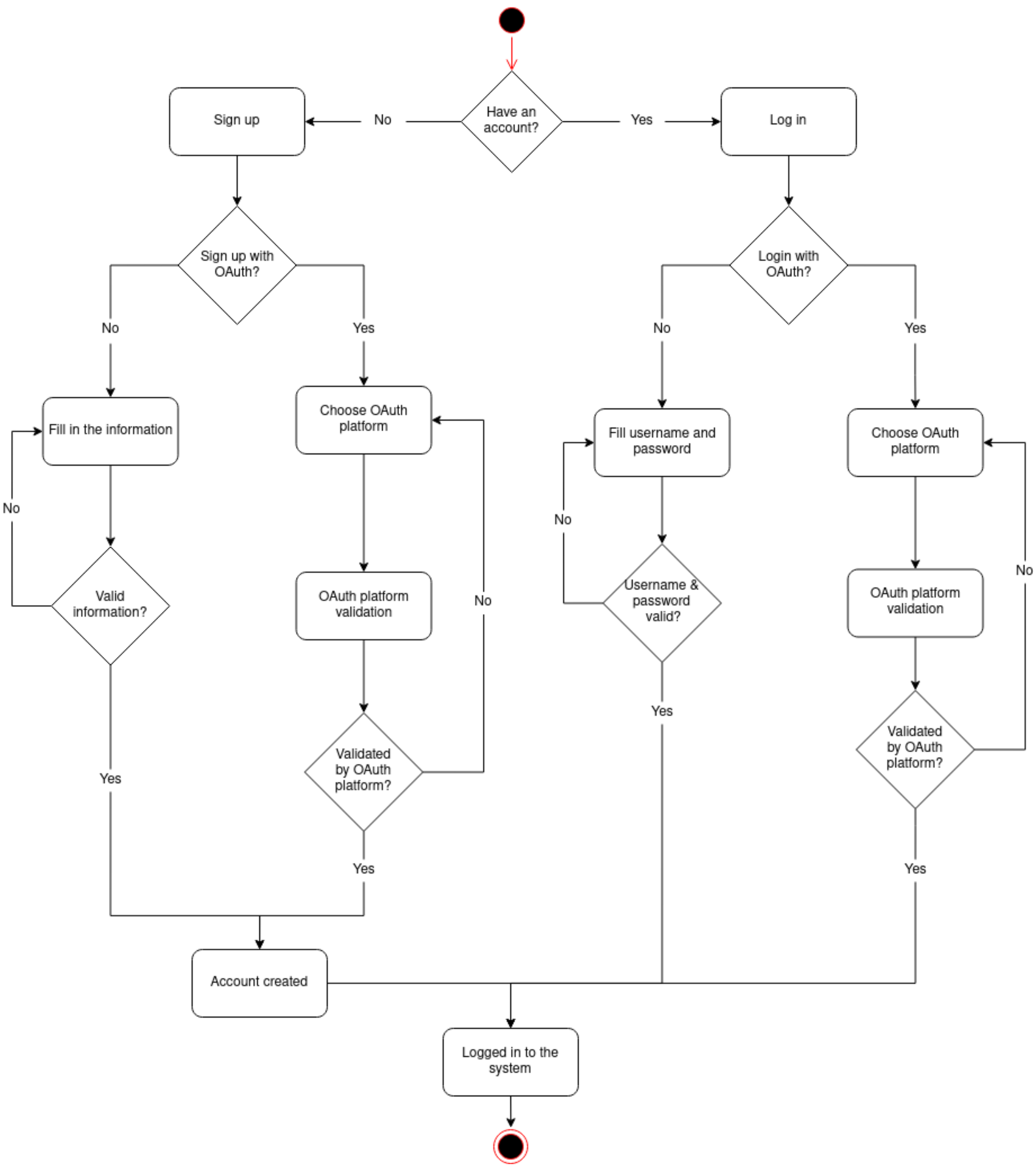


Figure 9. Log in and sign up flows

4.5.2 Profile

OAuth linking

It is also possible to link multiple OAuth platforms to one account (Figure 17). In that case, the user has to log in and head to the *"Profile"* page, where he can find the link under the *"Accounts"* tab. If an account from OAuth platform is already linked then the backend unlinks it from the account, otherwise the user is redirected to an appropriate OAuth platform for authentication. On success the user will be redirected to the frontend, and the user can access new repositories that this OAuth platform account has access to.

The backend also fetches all user emails from the linked account to later identify user commits. OAuth linking flow is described in Figure 10.

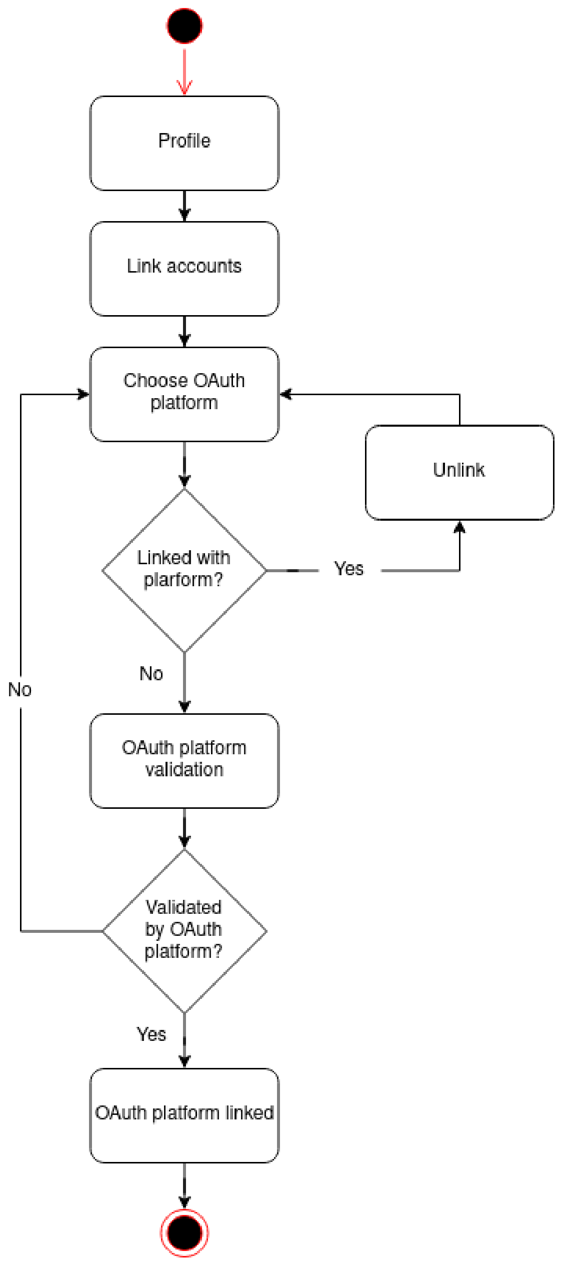


Figure 10. OAuth linking with user

Change password

Users can also change password (Figure 16). In the *"Profile"* view under the *"Change password"* tab user needs to fill in all the fields correctly for the password to be changed successfully. If the user does not have a password (OAuth registration), only new password fields have to be filled. Once a password is added, it can not be removed to avoid the possibility to having inaccessible accounts.

Repositories

Under *"Repositories"* tab user can see all his repositories (Figure 18). The user can start tracking a new repository by clicking *"Start tracking"* and then adding a webhook to the according OAuth platform. If *"Start tracking"* is clicked and everything is successful, a blue info box with instructions for webhook setup will pop up. In case the user wants to start tracking again then the repository can be found at the end of the list with a button *"Initialize again"*. Tracking can also be stopped by clicking on the red button with a trash can.

Delete account

Account deletion is under *"Delete account"* tab (Figure 19). To keep away accidental account deletion user has to write his name to the input box and then can proceed with the deletion. Unlinking all OAuth logins from account, that has no password login enabled is also considered as a deletion as otherwise it would become an orphan account. If the account is deleted, the user is redirected to the *"Login page"*.

4.5.3 Data visualization

Data visualization is divided into three pages - *"Dashboard"*, *"Leaderboard"* and *"Comparison"*. From the sidebar, the user can navigate to the necessary page, where on top of the page the selection of properties according to the graphs are seen.

Dashboard

"Dashboard" is the main landing page if the user is logged in. *"Dashboard"* view can be found in Figure 13. *"Dashboard"* contains three graphs and inputs. From the inputs, the user can change group, time period and interval. The user can choose between all the groups he has access to. The time period has a start, and an end which cannot be apart more than a year for performance reasons. Intervals are days, weeks and months.

The first graph is a combination of a line and a bar charts, which describes time spent writing code in a period of time and also shows how many users were active at the interval. The X-axis shows the period of time and Y-axis shows the time spent during the period. The Y-axis also shows active users count during the period. The line shows time spent and bar shows users count.

The second graph is a line chart, which describes activity hours during the chosen period. According to the interval, it shows daily activity, weekly activity or monthly activity. The

X-axis shows interval over selected time period, and the Y-axis shows an average time spent during each interval. If the interval is day then the X-axis describes each hour, in case it is a week, the X-axis describes each week day, etc. The Y-axis also shows lines added and removed from code. This graph brings out the time frame when users are actively programming.

The third graph is a bar chart that describes time spent in folders or files. The X-axis shows the time period, and the Y-axis shows the time spent during day/week/month depending on the chosen interval. Bars are stacked on each other and each bar describes one folder or file. This graph brings out which tasks are the most time-consuming and is very useful for lecturers, who want to see which homeworks take more time. The depth of how far into files tree the app goes to group files is set to 2, as this makes it usable for most of the subjects. It will be made configurable in the future.

Leaderboard

"Leaderboard" contains four inputs and two tables where users and files/folders can be easily compared. *"Leaderboard"* view can be found in Figure 14. Inputs contain group selection, date selection and depth selection. Depth determines how deep will the search go for files and folders.

The first table shows info about users in the chosen group. Data is divided into 7 columns - *"Total time"*, *"Commits"*, *"Lines added"*, *"Lines removed"*, *"Lines per hour"*, *"Commits per hour"* and *Lines per commit"*. By clicking on the column header the contents of the table will be sorted accordingly.

The second table shows info about the project(s) file system. Data is divided into 8 columns - *"Total time"*, *"Commits"*, *"Lines added"*, *"Lines removed"*, *"Lines per hour"*, *"Commits per hour"*, *"Lines per commit"* and *"Users"*. The sequence can also be changed by clicking on the column header. This table can be useful for comparing different tasks in university as they are usually placed in separate folders.

At the end of the page, there is an *"Export data"* button that allows downloading Comma Separated Values (CSV) file with data. The data is filtered by the same inputs that filter the data in tables but is grouped by also commits giving higher resolution than the tables described above.

Comparison

"Comparison" page has four inputs and one graph. *"Comparison"* view can be found in Figure 15. Inputs are group selection, date selection and interval selection. Multiple groups can be selected to compare them. Graph draws out line charts depending on the amount of groups selected. The X-axis shows the period of time and Y-axis shows time spent. Each line has different colour as it represents a different group and makes the graph more understandable. The graph can be toggled to show either total time or average time per user.

4.5.4 Licensing

It was decided in the beginning, that all the apps installed on the clients machine are made open source so that our users could inspect the code themselves and see that there are no malicious activities being done.

In the beginning, we planned to keep backend and frontend code as closed source. Later this plan was changed as we saw that Wakatime open source alternatives have received lots of features and updates during the development of our app and are really strong competitors. If we had decided to keep our app as a close source there is a high chance our users will turn to these solutions or build their own one.

To find suitable licence(s) for our apps, we took the top 3 most popular open source licences as of beginning of 2021 and decided to compare them to find which suits us the best. These three licences were Apache-2.0 Licence, Massachusetts Institute of Technology (MIT) Licence and GNU Lesser General Public License (GPLv3) according to the White source. As the top three contains both copy-left and permissive licences, we decided not to include any more licences in the comparison. [8] The results of comparison are show in table 13

	Apache-2.0	**MIT**	**GPLv3**
Market share	28%	26%	10%
Allows proprietary use	Yes	Yes	Yes
Allows source modifications	Yes, must state	Yes	Yes, must state
Allows sub-licensing	Yes	Yes	No
Disclose source	No	No	Yes

Table 13. Comparison of most popular open source licences.

MIT licence is the most permissive of these three as it only states that its copyright has to be included, and the author provides no warranties. MIT is also the shortest, and easily readable in it's wording. This makes MIT licence ideal for libraries that are supposed to be included in other applications but may cause no ambiguity in more complex cases. [9]

Apache 2.0 licence is similar to MIT as it's also a permissive licence. It differs from MIT licence by having more conditions clearly stated. Apache 2.0 licence also states that contributors provide an express grant of patent rights, meaning that it gives the user more protection about patent trolls. Apache 2.0 licence also states that changes made to source code have to be stated, but they can be licensed under different terms. [10]

GNU General Public License v3.0 on the other hand is one of the strongest copy left licences. It states, that if a piece of code licensed under GPLv3 is included in building some bigger application, then the source code of the whole application must be licensed under GPLv3 making the licence sticky. Although the licence states the code has to be publicly available, it does not prohibit commercial use. GPLv3 can be used to protect software against making it private. [11]

As the GPLv3's statement of making the whole source code public is really strong, we decided to also consider GNU Lesser General Public Licence v3.0 (LGPLv3) which is mostly the same as GPLv3, but is more permissive by stating that the source code does not have to be licensed under LGPLv3 if the original code is not modified and only used as a library. [12]

Licencing IDE plugins

It was decided to licence our IDE plugins under MIT licence. Our IDE plugins are relatively small and lack business logic in them. They are only required to integrate GTM into IDE-s, and we would greatly benefit if someone would make better alternatives as this would attract more users into using GTM. Also, most of, if not all the original Git-Time-Metric plugins are licensed under MIT licence and all Wakatime plugins are licensed under similarly permissive licences.

Licencing gtm-core

It was decided that similarly to IDE plugins, gtm-core is licensed under MIT licence. The biggest reason for it was that the original Git-Time-Metric app is licensed under MIT licence, and we have not made any huge changes. If we licensed our app under a more restrictive licence, users could easily turn to the original Git-Time-Metric app and develop it instead. As we have the same license as the original Git-Time-Metric app, there is a possibility that if there will be more work put into the original app than ours, we can merge our changes into the original app without having to change licensing.

Licencing gtm-sync

For gtm-sync it was decided to licence it under LGPLv3. As all the dependencies are licensed under MIT licence (many also dual-licensed under Apache 2 Licence), they are LGPLv3 compatible. As gtm-sync is pretty much a requirement to get data from Git notes to our back end, we want to make sure it, and its potential successors stay publicly available. We would like someone to base their closed source (and potentially non-free) application on gtm-sync. However, we do want to allow others to use gtm-sync as a library without requiring them to open source their project. If we allow users to use it as a library more freely, we also increase the chance to have other users contribute towards gtm-sync rather than their closed source fork or alternative.

Licencing gtm-api and gtm-front

We decided to licence gtm-api and gtm-front under the same licence as they are more closely tied together than other parts of the application. Similarly to gtm-sync all the dependencies are licensed under MIT licence (many also dual-licensed under Apache 2 Licence), they are LGPLv3 compatible.

For the licence we chose GNU Lesser General Public Licence v3.0. We strongly want these applications to stay open source also in the future, so a copy-left was logical choice. Although both of them are applications, not libraries we chose to use LGPLv3 instead of GPLv3, as technically some parts could be also used as libraries and that’s fine with us.

If someone wishes to use it without open sourcing their code, there is always an option to ask us as dual licensing is also a possibility. Then we can decide whether to allow it or not, but generally, we want the code to stay open source.

5. Results and validation

As a result of this project, a collection of applications was built to allow automatic time tracking for Git projects.

- Original Git-Time-Metric application was forked (named to gtm-core) and new features were added alongside numerous bug fixes.
- Jetbrains IDE plugin (gtm-jetbrains) was created to integrate gtm-core functionality into IDE and enable tracking without manual input.
- Gtm-sync client (gtm-sync) was built to safely fetch data with Git notes from Git, parse data from notes and sync up to backend.
- Web server backend (gtm-api) was built to persist data and serve it to the frontend.
- Web application frontend (gtm-front) was built to visualize data in an easy-to-understand format.

To validate the success of the project it is important to measure, how many students are using the application, how satisfied the students are with the application and how satisfied the lecturers are with the application.

Measuring amount of users using the app is the easiest, but the most important one. If there is a small number of users using the application then the application has failed its purpose and other aspects can be only used to analyse how to improve. Finding out the amount of users is rather easy, as the application itself stores this data, and the admin user can easily access it. In iti0201 course, we have time tracked from at least 34 different Git emails and in iti0202 we have data from at least 55 different git emails. As some people may have different Git emails on different computers, or they may change email during the course, we cannot say it is the same amount of students but guessing by emails we can say it is roughly 50% of students in these courses. We are very happy with this percentage of students already using the app as the application, especially the parts responsible for displaying data to users, were still in development throughout the course.

Although the number of users logically should reflect on their satisfaction in the application, it was still important to get more feedback from users to see what parts of the application need improvements. To get more feedback from students an anonymous questionnaire was created and sent to all students participating in iti0201, iti0202 and iti0301 courses in the spring semester of 2020/2021 study year. In the questionnaire, there were questions to users, non-users and users who had stopped using the application.

In total there were 18 responses, 13 were using GTM as of answering the survey, 4 had tried it but stopped using it and one respondent had not tried it. The person, who had decided not to try out the app stated, that he did not want to share his personal data with us and also brought out, that he believes measuring only time spent writing code is biased as it gives an incomplete picture about total time spent.

Out of the 4 users, who had tried the app but stopped using it, one said he stopped using the app as he has a more convenient alternative. Two respondents said they had some problems when using the app and did not feel like continuing to use it. One person said he did not like the idea of sharing such private data as time data and stopped using the app due to that. Out of these responses, it seems, that it is important to make the application usage more convenient to keep our users. We already have tweaked our application over the semester for easier usage, but there is definitely more room for improvements.

Out of the 13 people still using GTM, 9 said they are using it in either iti0202 or iti0201, and 5 that they are using in iti0301. These are the subjects that we and also lecturers endorsed students to use GTM at, so we were expecting most of our users to use it in one of these subjects. Additionally, 3 respondents said they are using GTM in other university subjects and another 3 were using GTM in personal projects. We are very happy that some people have been using the app also outside the subjects, where the lecturer endorses the usage of the app as it shows they like what the app is doing, and it solves some problem for them.

GTM users pointed out that they like the precision of time and how it is tied with commits and files, giving them an overview of what task and how long their time was spent on. Users also liked the web UI and how it has different ways to visualize the data both on the web and also on the command line. It was expressed that a big plus of the app is not requiring any manual input after initializing the tracking for the first time.

From the downsides, it was pointed out that viewing time data from the terminal is inconvenient for many users. It was also stated that the installation could have been easier and there could be a better web UI. Additionally, it was stated that the time widget in Jetbrains IDE-s sometimes stops working when multiple IDE-s are open which can be considered a bug.

The issue of having to do some things from the command line was actually solved in the latter half of the semester, but it seems we should have communicated it better as some people were not aware of it. Only slightly more than 50% of the users were using GTM web app with more than 30% never having visited it. Partially it might be caused by us not having the web UI ready for students, when we first introduced the app to them.

Some feedback was also collected from two lecturers, Ago Luberg and Gert Kanter, to get an insight into what could be improved for lecturers. Gert said he mainly looks at the total/average stats for his subject to have a general overview of how much time students are putting into the subject. He also brought out that per task statistics can be useful feedback when preparing tasks for the next year. These are the main areas we have also been focusing on, so we are very glad they overlap with what the lecturers are using.

Both Ago and Gert have asked, how can the university keep using the app without our assistance. We have some tutorials for setup, but there definitely is a need for a guide for a more advanced usage. There most likely will be some more features added during the summer to make the setup easier.

Both of the lecturers said the export data functionality is very useful, but Gert suggested allowing exporting the exact same data from tables that could be used without manipulation. This feature will be added before the start of the next semester.

For Ago there were some additional concerns, how the university can add features to the app after our graduation. As we decided to make all the applications open source, this is not so much of a problem anymore. We are glad to accept new features/bug fixes from other contributors, and we are glad if some other student from TalTech would continue with the project by connecting the application with Moodle and testing service.

Overall, the feedback was positive from both the students' and the lecturers. The application will likely be used in the future by the university and hopefully in students personal projects.

6. Comments

In this section, an overview is provided about the most impactful and changed technologies as well as the most important design choices.

6.1 Rust

Initially, we were thinking of choosing Java as our Backend language. As we started to analyze the needs, we ruled out C#, Python, C++ and choosing suitable language came down to Java and Rust. Both of us already had a lot of experience with Java and it is very widely used but on the other hand, the performance and lightweightness of Rust seemed to be over weighting Java. The easiest path would have been to choose already familiar Java, but we decided to go with Rust.

The main benefits of choosing Rust are

- Performance gain.
- It is lightweight.
- The benefit of learning a new programming language.
- Demonstrating fast adoption of new technologies which is super important in today's world.

The main cons of choosing Rust are

- Code writing efficiency is lower in the learning phase resulting in less functionality being made for the application.
- Significantly longer compile times than Java (and also longer than C/C++)
- Less optimal code than what it could be due to being inexperienced in language.

In the end, the decision to use Rust turned out to be brilliant. We had some struggles in the beginning, especially when working with dates and time, but they were not too hard to overcome. We were able to demonstrate the adoption of new technology while getting very good performance for our backend. Surprisingly the performance difference between Java and Rust can be noticed by a human without specifically measuring in some cases as

Rust has significantly faster cold start and faster response times. For bigger queries, this becomes less apparent depending on a case, as the most noticeable benefit comes from Rust having to do less Heap allocation, when responding to the request. Although we were afraid, that Rust code might be harder to write than Java code, that wasn't the case.

6.2 Changed technologies

The only change we made in the technologies was going from JavaScript to TypeScript in the frontend. JavaScript is the most widely used language for frontend projects, so we did not analyze the need for other technologies. As the project kept growing and the frontend started to get harder to read and organize, we came to the realization that a better solution is needed. Code readability is a very important part of this project as the school plans to use this application in the future and other students must understand the code quickly. We started to analyze the perks and cons of TypeScript (a superset of JavaScript) and turned out that it is a good solution for our problem.

Main benefits of TypeScript are

- It is strongly typed (helps with code readability and IDE support).
- Very similar to JavaScript.
- Both of us have previous experience with it.
- All the JavaScript libraries, and other JavaScript code works without any changes.
- Supports modules, generics and interfaces to define data.

The only debatable reason to use JavaScript is its performance gain over TypeScript. In our case, this does not play a huge role as the performance gain starts to shine in the hands of advanced JavaScript developers. Although we do suffer performance problems in some cases, the decision of TypeScript is not the root cause of the problem.

6.3 Using Git-Time-Metric as a base

Initially the Git-Time-Metric app seemed more pleasing to us as it already showed many fancy Git statistics whilst the main alternative Wakatime did not. We viewed some Wakatimes' open source alternatives but discarded them as they had only really basic features implemented meaning that we would have needed to build the backend on our own regardless of the choice of which app to use as a base. We went with Git-Time-Metric and we have not had any major setbacks, and thus we have proven this was a decent choice. However, there has been a lot of work put into also Wakatime's alternatives. If we needed to start again from scratch we would likely go with one of the Wakatime's alternatives instead as right now they already are decent open source backends.

7. Conclusion and future development

In conclusion, we are happy with the result as we have managed to solve the problem of measuring students time put into subjects for our customer. We managed to test our application out in two subjects and collect a reasonable amount of data.

It is very likely that the application will be used in full in the next semester in iti0102 course and hopefully also in iti0202, iti0201 and other similar programming subjects.

An avenue for future development is definitely integrating GTM to Charon/Moodle and tester as it allows more convenient usage for lecturers. As linking the data with test results gives an opportunity to do some machine learning we hope someone can do it for their final thesis in the future.

As we are planning to continue using the app personally we are also planning to put more work into it in the summer to also improve the GTM web application usage for personal projects.

For the web UI the main ideas for improvements are:

- Add separate view for viewing user personal data over all repositories.
- Add branch/user-based filtering to the *"Comparison"* tab and also add some statistics to it in numeric format.
- Allow students to view their ranking in *"Leaderboard"*.
- Allow users to create their own groups for better organization.
- Improve charts performance as there is some lag for longer periods of time.
- Move install instructions to our UI.
- Allow exporting data from *"Leaderboard"* tables.

Bibliography

[1] Roxine Kee. *The Best Time Tracking Apps in 2021: Toggl, Clockify, and More*. [Accessed: 05-05-2021]. URL: https://collegeinfogeek.com/time-tracking-app/.

[2] Régis Freyd. *Support time logging in commit messages*. [Accessed: 14-04-2021]. URL: https://gitlab.com/gitlab-org/gitlab/-/issues/16543.

[3] *Domain driven design*. [Accessed: 04-05-2021]. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html.

[4] *State of frontend*. [Accessed: 04-05-2021]. URL: https://tsh.io/state-of-frontend/#frameworks.

[5] *Recarts*. [Accessed: 14-04-2021]. URL: https://github.com/recharts/recharts.

[6] Jeb Rosen. *rocket_oauth2*. [Accessed: 4-04-2021]. URL: https://docs.rs/rocket_oauth2/0.4.1/rocket_oauth2/.

[7] Internet Engineering Task Force. *The OAuth 2.0 Authorization Framework*. [Accessed: 4-04-2021]. URL: https://tools.ietf.org/html/rfc6749.

[8] Patricia Johnson. *Open Source Licenses in 2021: Trends and Predictions*. [Accessed: 07-05-2021]. 2021. URL: https://www.whitesourcesoftware.com/resources/blog/open-source-licenses-trends-and-predictions.

[9] *MIT License*. [Accessed: 07-05-2021]. URL: https://choosealicense.com/licenses/mit/.

[10] *Apache License 2.0*. [Accessed: 07-05-2021]. URL: https://choosealicense.com/licenses/apache-2.0/.

[11] *GNU General Public License v3.0*. [Accessed: 07-05-2021]. URL: https://choosealicense.com/licenses/gpl-3.0/.

[12] *GNU Lesser General Public License v3.0*. [Accessed: 07-05-2021]. URL: https://choosealicense.com/licenses/lgpl-3.0/.

Appendices

Appendix 1 - Project links

Item	**Link**
gtm-api source code	`https://github.com/DEVELOPEST/gtm-api`
gtm-front source code	`https://github.com/DEVELOPEST/gtm-front`
gtm-sync source code	`https://github.com/DEVELOPEST/gtm-sync`
gtm-core source code	`https://github.com/DEVELOPEST/gtm-core`
gtm-jetbrains source code	`https://github.com/DEVELOPEST/gtm-jetbrains`
gtm-vim source code	`https://github.com/DEVELOPEST/gtm-vim`
Main issues	`https://github.com/DEVELOPEST/gtm/issues`
gtm-core issues (used in early stages of development)	`https://github.com/DEVELOPEST/gtm-core/issues`
Deployed front end	`https://cs.ttu.ee/services/gtm/front/`
Jetbrains plugin	`https://plugins.jetbrains.com/plugin/15794-gtm-enhanced`
Install instructions	`https://gtm.pages.taltech.ee/install/`
Openapi documentation for backend	`https://cs.ttu.ee/services/gtm/api/swagger/index.html`

Table 14. Project links

Appendix 2 - Frontend views

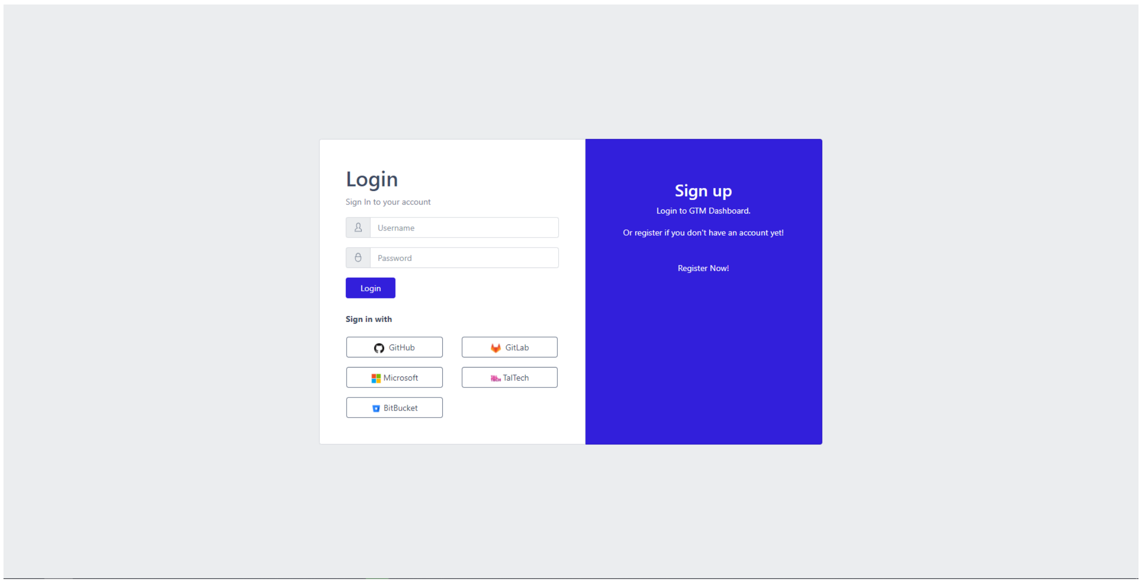


Figure 11. *Login view.*

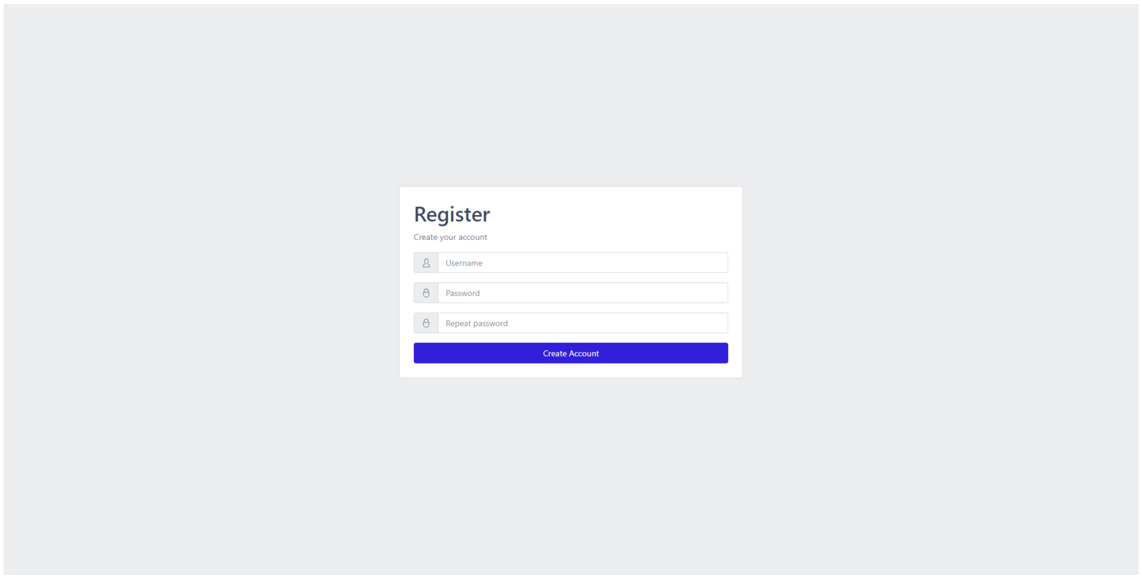
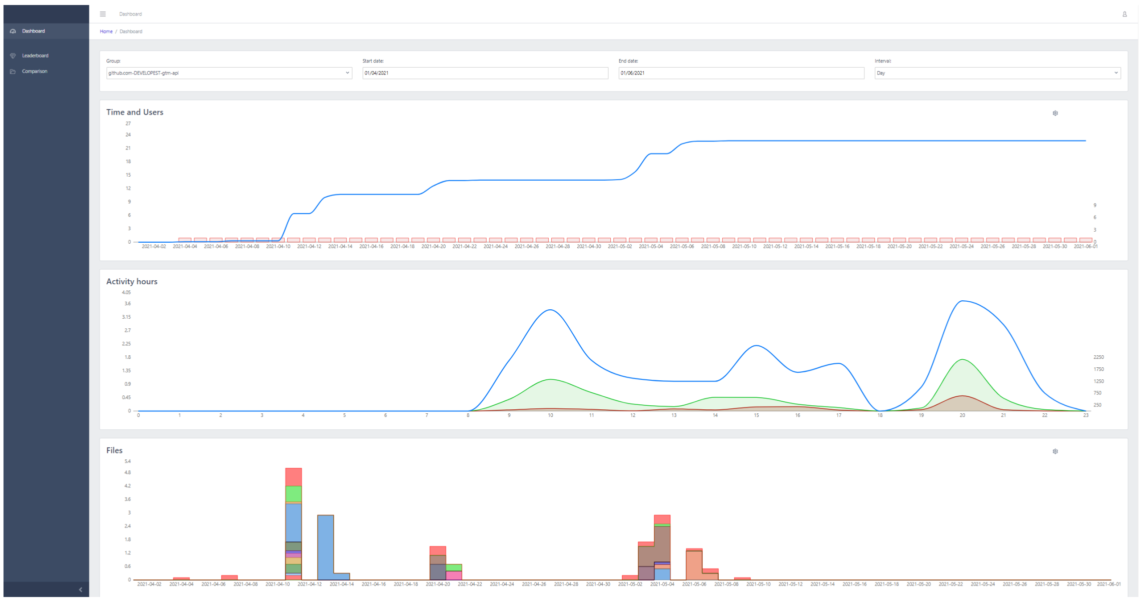
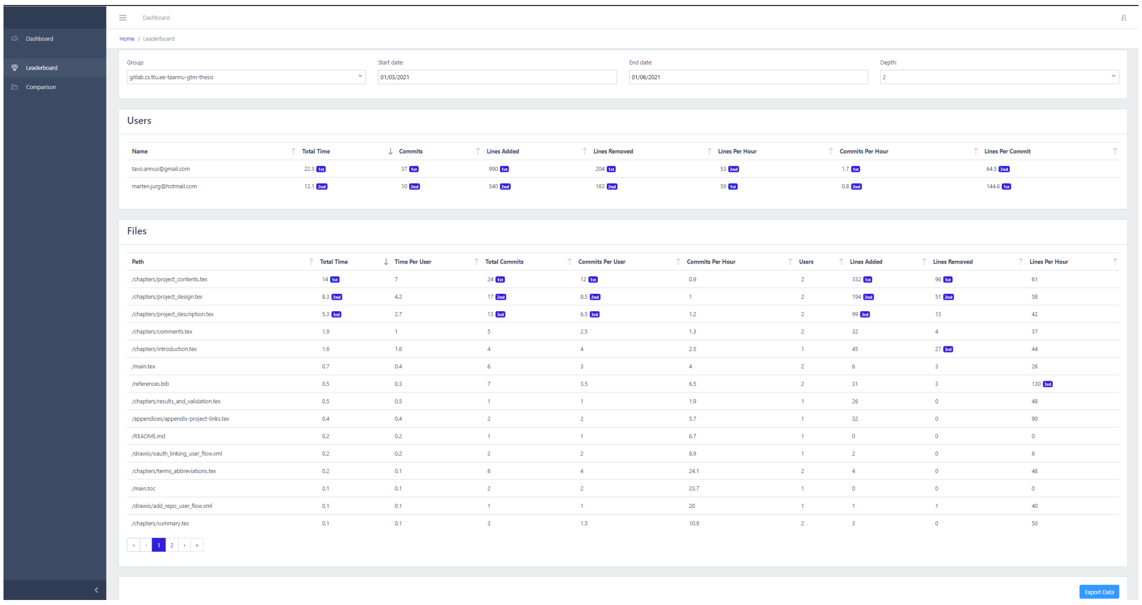


Figure 12. Register view.



Group: github.com-DEVELOPEST-gtm-api
Start date: 01/04/2021
End date: 01/06/2021
Interval: Day

Figure 13. Dashboard view.



Group: gitlab.cs.ttu.ee-taannu-gtm-thesis
Start date: 01/03/2021
End date: 01/06/2021
Depth: 2

Users

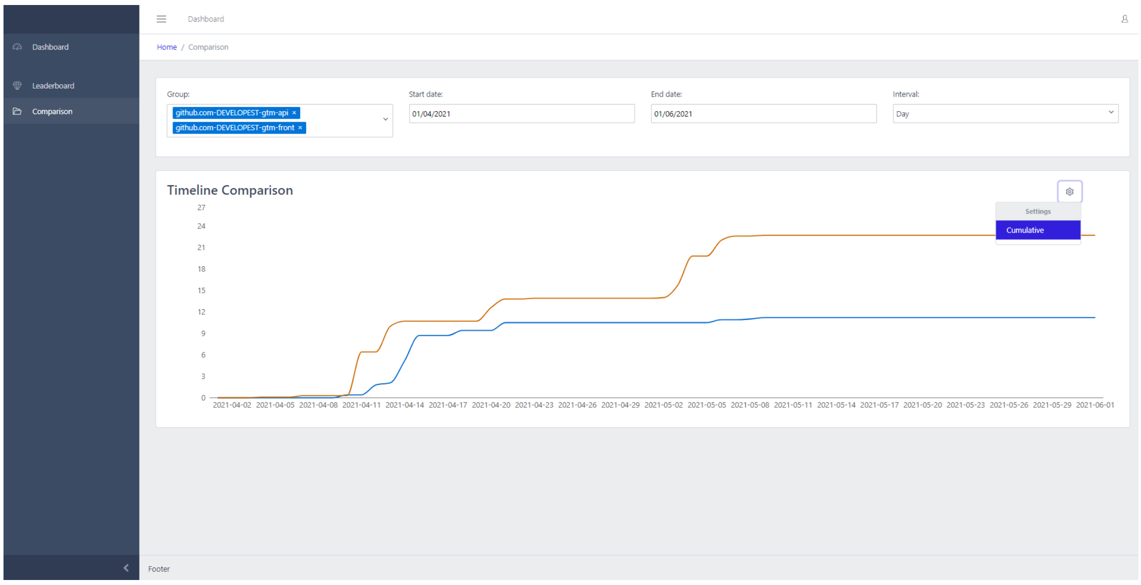
Name	Total Time	Commits	Lines Added	Lines Removed	Lines Per Hour	Commits Per Hour	Lines Per Commit
tavo.annus@gmail.com	22.3 1st	37 1st	990 1st	204 1st	53 2nd	1.7 1st	64.5 2nd
marten.jurg@hotmail.com	12.1 2nd	10 2nd	540 2nd	183 2nd	59 1st	0.8 2nd	144.6 1st

Files

Path	Total Time	Time Per User	Total Commits	Commits Per User	Commits Per Hour	Users	Lines Added	Lines Removed	Lines Per Hour
/chapters/project_contents.tex	14 1st	7	24 1st	12 1st	0.9	2	332 1st	96 1st	61
/chapters/project_design.tex	8.3 2nd	4.2	17 2nd	8.5 2nd	1	2	194 2nd	51 2nd	58
/chapters/project_description.tex	5.3 3rd	2.7	13 3rd	6.5 3rd	1.2	2	99 3rd	13	42
/chapters/comments.tex	1.9	1	5	2.5	1.3	2	32	4	37
/chapters/introduction.tex	1.6	1.6	4	4	2.5	1	45	27 3rd	44
/main.tex	0.7	0.4	6	3	4	2	6	3	26
/references.bib	0.5	0.3	7	3.5	6.5	2	31	3	130 3rd
/chapters/results_and_validation.tex	0.5	0.5	1	1	1.9	1	26	0	48
/appendices/appendix-project-links.tex	0.4	0.4	2	2	5.7	1	32	0	90
/README.md	0.2	0.2	1	1	6.7	1	0	0	0
/drawio/oauth_linking_user_flow.xml	0.2	0.2	2	2	8.9	1	2	0	8
/chapters/terms_abbreviations.tex	0.2	0.1	8	4	24.1	2	4	0	48
/main.toc	0.1	0.1	2	2	23.7	1	0	0	0
/drawio/add_repo_user_flow.xml	0.1	0.1	1	1	20	1	1	1	40
/chapters/summary.tex	0.1	0.1	3	1.5	10.9	2	3	0	50

« ‹ 1 2 › »

Figure 14. Leaderboard view.



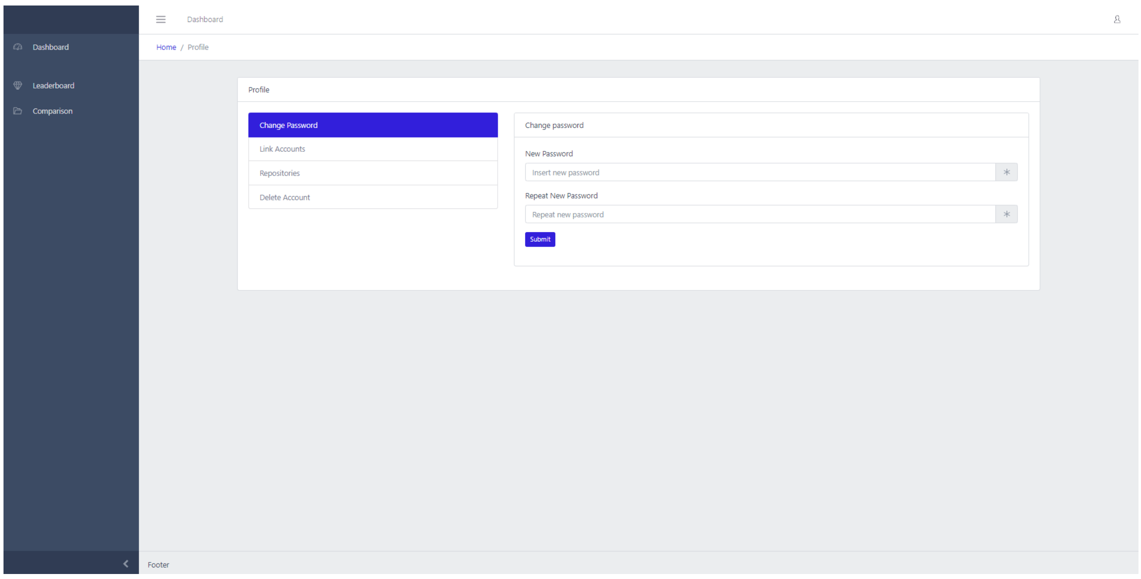
Group: github.com-DEVELOPEST-gtm-api × github.com-DEVELOPEST-gtm-front ×

Start date: 01/04/2021

End date: 01/06/2021

Interval: Day

Figure 15. Comparison view.



New Password: Insert new password

Repeat New Password: Repeat new password

Figure 16. Change password view.

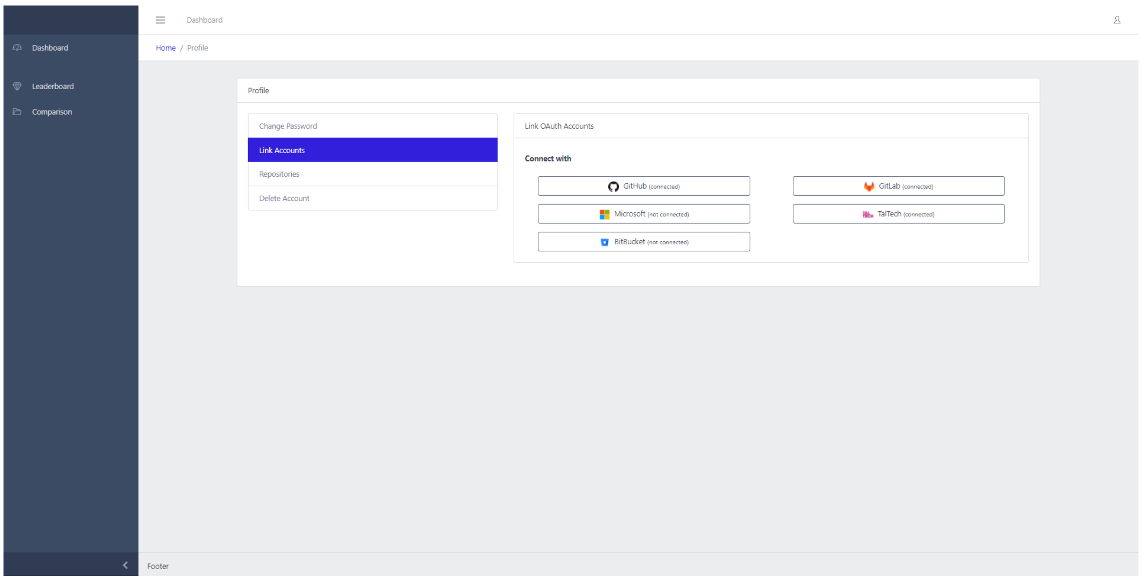


Figure 17. Link accounts view.

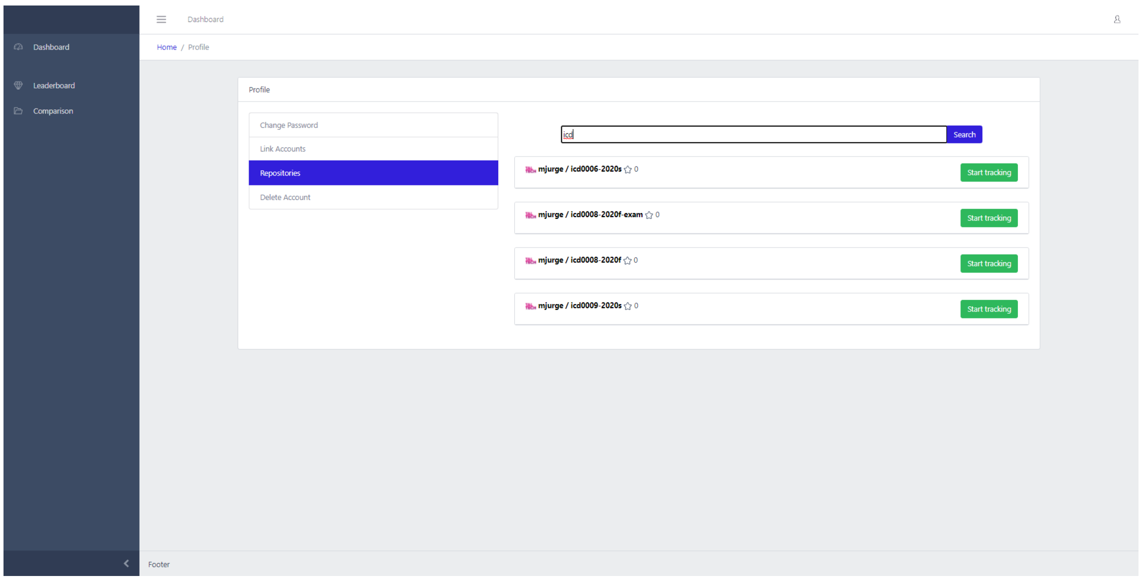
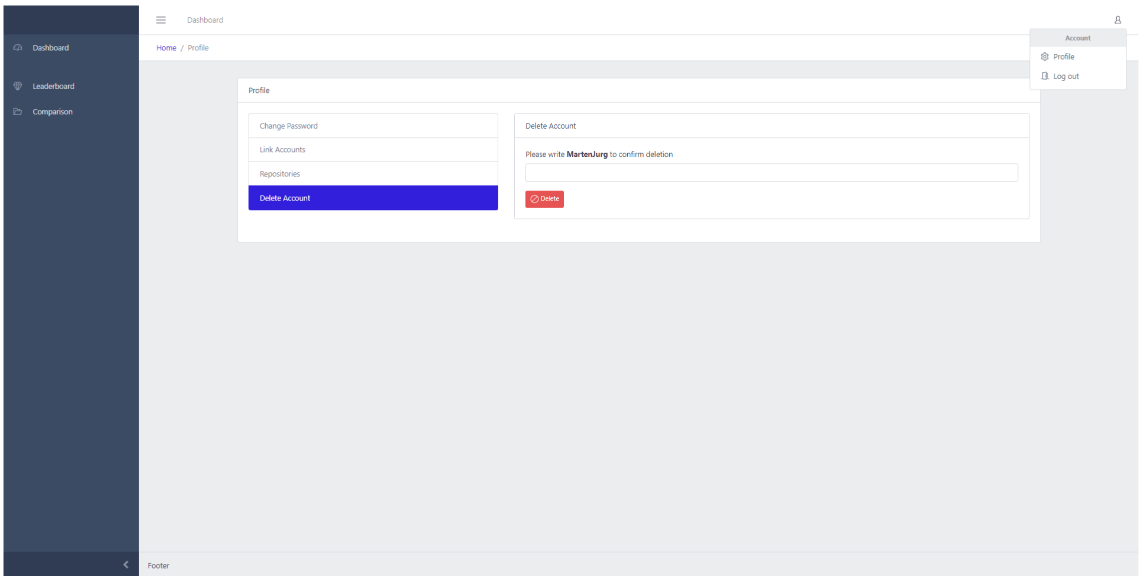


Figure 18. Repositories view.



Please write **MartenJurg** to confirm deletion

Figure 19. Delete account view.