FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Git-based Wiki System
Student:	Bc. Jaroslav Šmolík
Supervisor:	Ing. Jakub Jirůtka
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

The goal of this thesis is to create a wiki system suitable for community (software) projects, focused on technically oriented users. The system must meet the following requirements:

- All data is stored in a Git repository.
- System provides access control.
- System supports AsciiDoc and Markdown, it is extensible for other markup languages.
- Full-featured user access via Git and CLI is provided.
- System includes a web interface for wiki browsing and management. Its editor works with raw markup and offers syntax highlighting, live preview and interactive UI for selected elements (e.g. image insertion).

Proceed in the following manner:

1. Compare and analyse the most popular F/OSS wiki systems with regard to the given criteria.
2. Design the system, perform usability testing.
3. Implement the system in JavaScript. Source code must be reasonably documented and covered with automatic tests.
4. Create a user manual and deployment instructions.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 3, 2018

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Git-based Wiki System

Bc. Jaroslav Šmolík

Supervisor: Ing. Jakub Jirůtka

10th May 2018

Acknowledgements

I would like to thank my supervisor Ing. Jakub Jirutka for his everlasting interest in the thesis, his punctual constructive feedback and for guiding me, when I found myself in the need for the words of wisdom and experience. My heartfelt gratitude belongs to my brother, Ing. Jiří Šmolík for his endless patience, dedication and reviews of the text in every stage of development. I am also grateful to my colleagues: Peter Uhnák, Maroš Špak and Petr Chmelař for cooperation on the project of designing the user interface of the editor, as well as to all who took part in the usability testing. Last but not least, I would like to profusely thank 9gag, for helping me to find joy in the troubled times of dismay.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 10th May 2018 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Šmolík, Jaroslav. *Git-based Wiki System.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem práce je vyvinout wiki systém založený na Git, s podporou jednoduchých značkovacích jazyků a řízení přístupových práv. Systém je při analýze definován standardními prostředky. Vzhledem k jeho požadavkům jsou zhodnoceny existující open-source wiki systémy. Řešení jejich slabin je diskutováno společně s návrhem systému, dle kterého je implementován. Závěrem je popsáno testování, jak automatické, tak prostředky testování použitelnosti.

Klíčová slova wiki systém, webová aplikace, Git, jednoduché značkovací jazyky, JavaScript, Node.js

Abstract

The goal of the thesis is to develop an access-controlled wiki system based on Git, with the support of lightweight markup languages. In analysis, the system is defined through the conventional methods. With regard to its requirements, existing open-source wiki systems are reviewed. Solution to their weaknesses are discussed along with the system's design, according to which it is implemented. Finally the automatic testing and the results of the usability testing is commented on.

Keywords wiki system, web application, Git, lightweight markup languages, JavaScript, Node.js

Contents

List of Figures

List of Listings

Introduction

There is a large variety of specialized tools that can be used for managing wikis. Most of the systems however, are focused on the ease of use for the new or unexperienced users. They lack features desired by the advanced users, such as software developers. The thesis tackles to solve the issue by developing a wiki system for the unexperienced and the advanced users alike, to provide an interface for developing complex, structured documents.

The means of achieving the goals, namely the potential to satisfy the advanced users are of architectural nature. They include, but are not limited to, the incorporation of years tested technologies used by developers, to design the system upon. One of which is Git, which is a powerful VCS, growing in popularity amongst OSS community and even corporate development. Another example is an emphasis on using a LML for the document notation. LMLs, such as Markdown or AsciiDoc are favored over binary document formats for their readability and line-oriented snapshot versioning potential; and even over the complex and difficult to write markup languages (e.g. LaTeX or HTML), for their simplicity. A strong aspect of the system supporting efficiency and comfort of use for the advanced users is to provide a CLI via Git, apart from a WUI. The two interfaces provide equal access options for the users. For better support of the collaborative development, the system features access control.

In *the following chapter* the assignment instructions are elaborated and it is explained what is considered a *wiki system*.

A high abstraction concept of the system from the user perspective is demonstrated on the business process model in the *analysis*. The following section in the chapter (*User access*), resolves fundamental architectural decisions regarding authorization, which force restrictions on the system requirements. Finally the system is defined through the conventional means of requirements and use case model and then the relations between the individual use cases and the functional requirements are commented on.

With the system definition from the analysis, the existing wiki systems that use the Git VCS are researched and reviewed from the perspective of criteria defined in the analysis. Doing which, the defects of the systems regarding the criteria in the summary are pointed out.

Solutions for the discovered imperfections in the systems are proposed in the chapter *Design*. Apart from that, the chapter discusses the major libraries used in the system that have impact on its design. The architecture of the

system is presented and description of its core components is provided. A low fidelity prototype of the UI is designed.

In the following chapter *UI testing* a static usability testing using a UI heuristic is performed, the results and proposed corrections for the identified issues are presented.

The implementation information is briefly summarized in the chapter *Implementation* and selected obstacles faced during the realization are discussed along with the applied solution.

The technologies used for the automated testing along with the materials, process and the conclusion of the usability testing are included in the final chapter *Testing*.

Chapter 1

Thesis' goal

The target of the thesis is to explore the state of the art of version controlled wiki systems, find the most appropriate solution for the design and implement a system with the special features, which vaguely described in this section and fully specified in the analysis.

1.1 What is a *wiki*?

The system is a version controlled, document oriented, text-based wiki software.

But what is a *wiki*? According to the Oxford dictionaries it is a *"a website that allows any user to change or add to the information it contains"* [73] a longer and certainly more entertaining description from WikiWikiWeb[1] states: *"The idea of a Wiki may seem odd at first, but dive in, explore its links and it will soon seem familiar. Wiki is a composition system; it's a discussion medium; it's a repository; it's a mail system; it's a tool for collaboration. We don't know quite what it is, but we do know it's a fun way to communicate asynchronously across the network."*[2] [92]. Those are both obviously vague concepts, but set up a reasonable foundation of what wiki is.

The essential idea behind wiki has always been to put the users into the role of not just mere consumers, but producers of the content, which is the true feature of *Web 2.0*. Wiki is a platform for document oriented, organically created structured content.

Documents are typically managed by moderators (who perform corrections, topic creation etc.) and written by a community of users. The community users have an easy access for the document editing and thus it is easy for them to become a contributor. An in-browser solution for the article editing is expected, for users to collaborate, because if users feel uncomfortable, or face a steep learning curve, they might become discouraged from their participation and then the wiki cannot thrive without content updates.

Apart from the editing interface, it is expected of a wiki to provide documents of a common topic, that binds the community of contributers. *Wikipedia*

[1] WikiWikiWeb is often considered to be the first wiki being launched in 1995. [94]

[2] The insecurity in the description (*"We don't know quite what it is, (…)"*) is understandable from WikiWikiWeb, since its creator chose a name and needed to explain the concept to visitors at the time, when it was not entirely sure, how the system is going to evolve.

[88] is an encyclopedia, *WikiWikiWeb* [93] mentioned earlier, was created to discuss design patterns, or *Wiktionary* [90] as a community developed multilingual open dictionary are all fine examples of such trait.

1.1.1 Side-note on wiki topic

It is convenient at this point to clarify, whether there is a topic to Gitwiki.

Gitwiki is not a *wiki* as Wikipedia. It is a *wiki software* which can be used to create a topic oriented *wikis*. A great example of a wiki software is GitHub Wikis [29], for instance. It is a part of GitHub web application that allows users to provide e.g. user manuals for software repositories etc. GitHub Wikis have no topic though it is expected to be used to create an elaborate software documentation, for a specific software. The resulting product is a *wiki* as it was discussed. It is bound to a specific topic by its contents.

1.2 Real world usage of the system

While the target system is potentially a universal document management platform, usable for a wide variety of applications, an example archetypal usage scenario is set. The system is considered (for the purpose of design decision making, interface design, etc.) a platform for an API documentation (or a user manual) collaborative creation.

This defines the special traits of the system, which make it distinguish itself amongst others.

Having mentioned the general potential of the system, other applications the system should satisfy with minor effort would be:

- a publishing platform,
- a collaborative maintenance of large documents that are too large for online services such as Google Docs [39] or Microsoft Word Online [66],
- a students' hub,
- or a university's tool for the writing and submission of final theses.

1.3 Distinctive features

1.3.1 Emphasize lightweight markup languages

Markdown has become more or less a standard for the *readme* files and documentation. It is easy to learn; intuitive to read, even if you are not familiar with the syntax; machine readable and it has many tools for comfortable writing for users familiar with RTEs. Once familiar with the basic syntax, it is not an issue to write documents in a simple text editor.

Most of these features, though not necessarily all, are typical for most LMLs. A form of a simpler markup language is used in almost every wiki system. The reason to favor Markdown [40] and AsciiDoc [5] over Wikitext [89], used by MediaWiki for example, is that the former are rooted in the developer community. LMLs bring advantage of familiar syntax to users for the archetypal usage as well as the prioritized special syntax features for development, such as source code snippets etc.

1.3.2 Focus on advanced users

As already mentioned, easy to use interface is a core feature of a wiki system. This affects the UI design of a wiki system. It must be welcoming to new users. The UI must be fool-proof, forgiving and guide the user through editing process.

The documentation platform is not like that. Its users write often. It is part of their job and they know the document syntax by heart. They do not need the system to slow them down by clicking formatting buttons for the few formatting options e.g. Markdown has.

The system should by all means provide an intuitive interface, but not at the cost of the use efficiency for the advanced users.[3]

1.3.3 Use robust non-linear VCS

The raised scenario requires to be able to track changes and their authors as well as to be able to return to the previous revision. This feature is fairly common and almost required for all wiki software, because the opened collaboration might be abused by attackers or vandals.

Developing parallel versions of the content is fitting for the scenario. This feature could be used to be able to maintain API documentation for distinct versions of the software, for instance. The technique is usually called "feature branch" in the software development. The developer creates new features in separate version branches, detached from the master branch. This allows to add the feature as a single atomic revision, when it is properly tested.

1.3.4 Provide direct access to the repository

Many wiki systems provide access only via WUI. This might be sufficient in many cases, when the only presentation of the content is on the web in the exactly same form. It is a common practice that the user manuals are linked together into a single large document, that is provided as a whole in a printable or online format. It is not important what tools are used to do so, but rather to provide the user (or their script or plug-in) with the access to the files directly.

This access should be provided for the read as well as the write operations, to allow users to edit the repository in their own environment.

[3]This does not mean that the UI should be unintuitive for new users, but rather it should focus on advanced features allowing the experienced user a swift interaction resembling the one they know from IDEs.

Chapter 2

Analysis

2.1 Business process model

In this section, abstract needs of the users in form of a business process model are formalized. It provides necessary data for the system requirements specification.

As stated before, the system at hand has a potential to be far more than a documentation platform. For the discussion of the users' needs and preferences, the mentioned referential application is used throughout the thesis.

Here in business process model. The workflow of a company using the system is described. While taking a very specific direction, the example case is convenient for two main reasons:

1. it portraits usage of the system by software developers, who are example of *technically oriented users*, as stated in the thesis assignment instructions,
2. it is elaborate enough to demonstrate the several user types with different needs and expectations of the system. This helps to model it and its variability.

In the scenarios it is assumed, a part of the development team is working on a user manual for their software product. The team consists of:

- the head of the department, who takes care of the project management of the user manual development,
- several developers, who are familiar with their software module and each write a manual for their code,
- a reviewer, who fixes typos, grammar, stylistics etc. and
- a publisher who takes the source codes and produces the final manual for DTP and print purposes.

The department is using Markdown, since all the developers who are writing the most of the texts are familiar with it and use it efficiently.

In the following two subsections the interaction of the development team with the system from business process perspective is demonstrated.

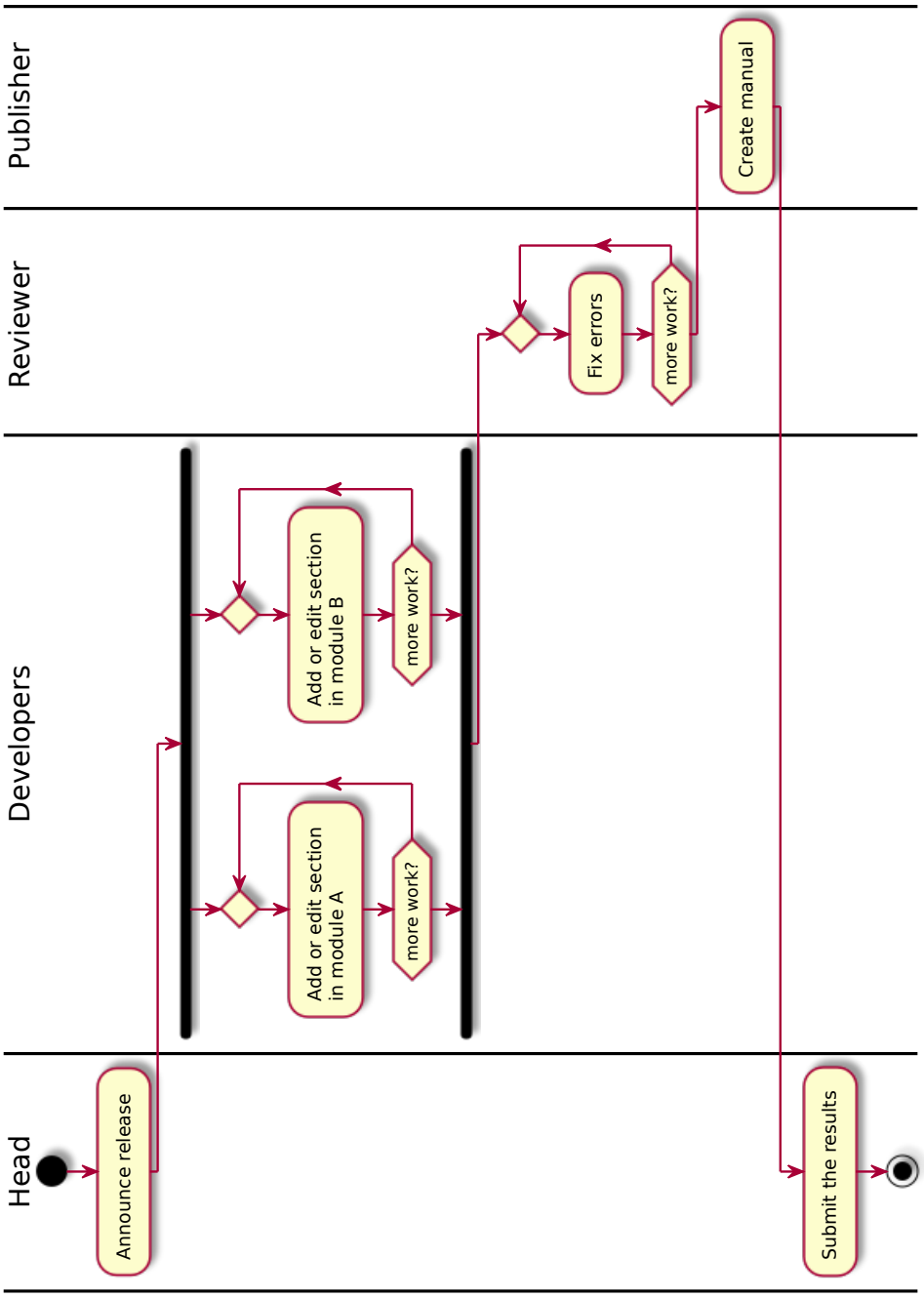

Figure 2.1: Business process model: Release

2.1.1 Business process scenario: Release

In the diagram 2.1, there is a possible scenario of the department's workflow on updating user manual after software release. The head of the department orchestrates the team to start working on the manual for the new version. The two developers keep updating the manual pages, each for their module. The pages are reviewed by the reviewer. When they are satisfied with the text, it is submitted to the publisher, who bundles the pages and create a single readable and printable document, which is submitted back to the head of the department.

A few observations from the diagram are made:

- If a logical mistake appears in the final product, the head might want to know who caused it. The head wants to review the content the same way a source code can be reviewed, providing access for the individual annotated lines, containing the revision ID, date and author information. VCS and revision updates are required.
- The publisher needs a direct access to the files.
- The first developer only writes in pages regarding module A, while the second for module B. The publisher only needs a read access to the files. Potential mistakes can be avoided, if the VCS repository was divided into namespaces and edited with access control management.

2.1.2 Business process scenario: Hotfix

The diagram 2.2 showcases the department's flow of action, when a hotfix, which requires an update of the user manual, is issued. The process is very similar to the previous scenario, because the processes are discussed at a very high level of abstraction. The notable difference however, is that multiple maintained versions of the manual need to be accessed and updated separately. The following observation are formed: If a hotfix that requires a change in the user manual is created, it is demanded to patch the previous version of the manual, without introducing the changes that are already applied to the current version of the manual – In the same manner as changes to source code are applied. This calls for a VCS with the parallel branch support.

2.1.3 Summary

Notable conclusions from the observations are as follows:

1. The system *needs* an underlaying VCS with a branching feature, as pointed out by several observations. A distributed VCS is utilized in order to allow participants to make contributions when out of the office.
2. The system *needs* to provide a direct access to the files. At least the read permission for the publisher.
3. The system *should* provide a direct access to the files for the revision updates as well, since the developers likely work in their own environment most of the time, at their personalized workstations.
4. The system *should* provide a simple read/write interface with no setup required for the reviewer or head of the department. Both participants

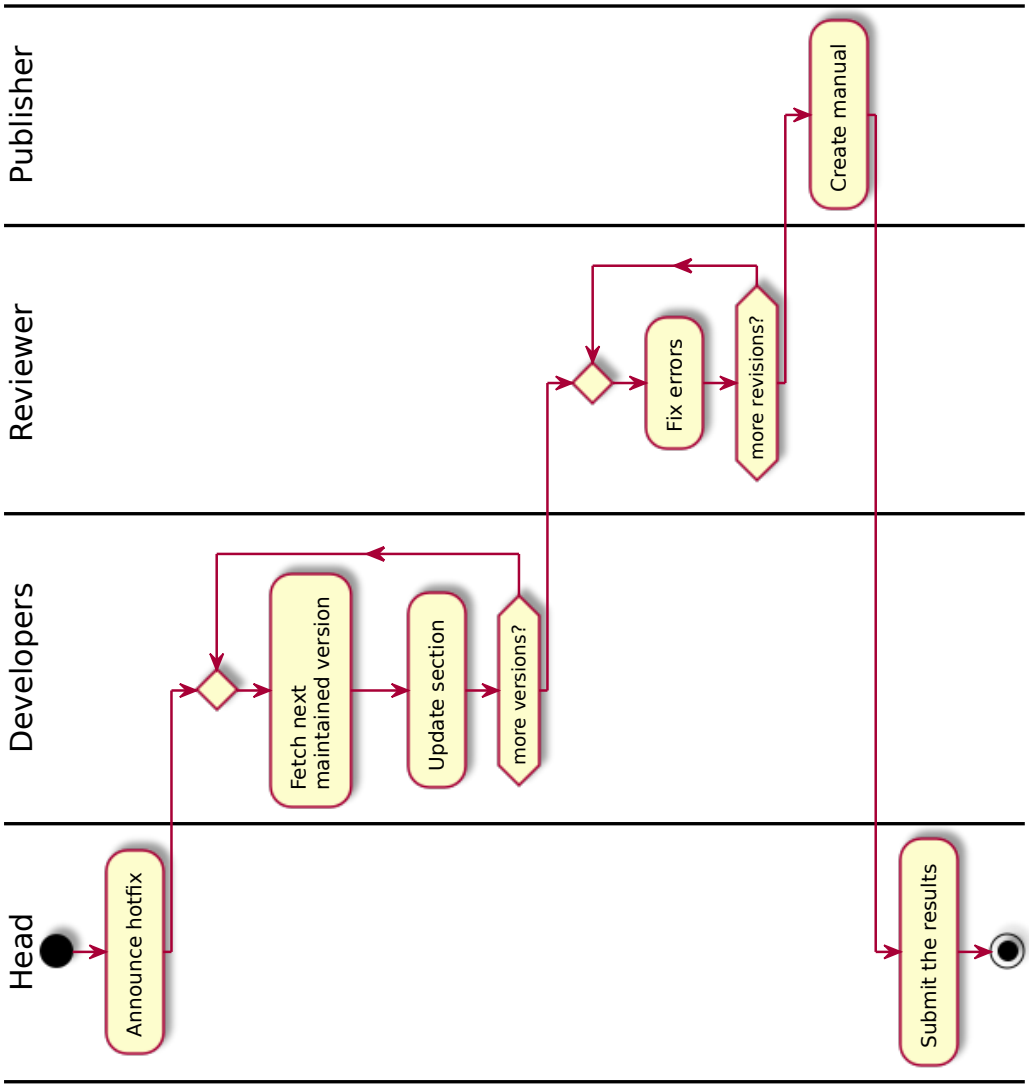

Figure 2.2: Business process model: Hotfix

are then able to preview the pages in a more familar form compared to the Markdown source code.
5. The system *should* provide an in-repository access control, to prevent unintended revisions.

2.2 User analysis

In the previous section the business processes of a small department, developing a user manual for their software were mapped.

Hence firm decisions (with fatal impacts on the system, including its requirements) are to be made, archetypal users mentioned in business process model are discussed.

This section is a brief user persona definition. It is intentionally drawn back from the lengthly formal persona definition from the usability testing [85], since a detailed persona definition proves less useful for a one man team.

The users are described with regard to their role and their knowledge and skills regarding the system. They are given names for further convenience of the thesis, but the part of persona definition regarding their personal life and the related features of the archetype personification is neglected.

The head of department is a lady called Hump and she just organizes the team. Her job related to the system is very limited, but includes potential access control setting. She is also a developer. In a scenario where Hump manages several teams access rights, it might be useful to keep the settings in a configuration file with the scripting potential.

The developers are programmer gentlemen called Dump and Lump. They are working in an IDE or coding editor and they are very efficient using it. It is best to let them work in their natural environment. Dump and Lump know the selected LML by heart and they are skilled at reading it from the source code, as well as writing it without additional visualization or preview tools. They are familiar with core VCS principals and use a VCS on their daily bases, when coding in the team.

Reviewer Rump is not a developer and just checks for readability, typos, etc. He makes subtle changes in the manual. Rump knows the LML syntax, but prefers an easy to use tool with a rendered preview to read the texts in a formatted document. He knows of VCS and its basic principles, but does not use it often.

The publisher named Pump makes no changes to the pages and just downloads the repository to his workstation and produces the desired outputs using a set of maintained scripts.

2.3 User access

As mentioned in the previous section, there are issues that need to be resolved, before approaching the rest of the analysis. Because the impact of the resolutions, it is required to completed even before defining the system through standard tools, such as the requirement model. That is the way the documents are persisted, and how the users access it.

To this moment in order to remain at the abstract conceptual level, which was convenient for e.g. business process modeling, it has not been implied a

specific VCS is used within the system, though it is stated in thesis assignment instructions. Git VCS is used for wiki contents persistence, as instructed. It has many advantages, including a branch model, a CLI repository access via SSH and it is decentralized, which is convenient for Dump and Lump (the developers) when working out of the office, as pointed out by an observation in the business process model. Apart from that, it is fairly popular. According to [8] up to 50% of the existing open source projects use Git, while the second place goes to Subversion with 42%. This applies only for the open source projects. Most of the statistics reflecting the global usage of VCSs are thus misleading, and in global scope, with the private projects included, Subversion plausibly still rules over Git with usage statistics. From various sources, e.g. mentioned in [64], it is apparent that Git's popularity is increasing over the years nevertheless, which makes Git a reasonable choice.

Using Git as an underlaying VCS layer brings two important questions to discuss.

1. Access control gets more complicated. In a centralized VCSs, it is natural to have the feature of file locks, which is e.g. available in Subversion. Though there are tools to simulate this in Git, it becomes far more challenging. How is the access control within a Git repository solved?

2. Git provides a useful interface for repository cloning, granting an elegant solution for the direct file access, familiar to Dump, Lump and Pump. Users are authenticated through an SSH authentication layer, once they deliver their public keys to the hosting server. This is a standard practice used by the popular Git hosting providers. How is the user authenticated and how is the identity paired with the stored public key?

2.3.1 Authorization

A tool for an authorization layer atop the SSH to manage Git repository access for the Git hosting is required. The possible open source options available are discussed.

There are many Git hosting services with a swarm of supportive features, such as the code review, issue tracking and even access control. These self-hosted services include e.g. GitBucket [82], GitLab [33] or Gogs [36]. None of the services are suitable, since none of them by this time offer a modular usage, to utilize just the mere SSH authorization layer. GitLab is selected from the group to demonstrate this.

Since none of the examples from the said group are convenient[4], software that serves only authorization purpose is discussed. There are two examples: Gitorious [81] and Gitolite [15]. Gitorious is no longer maintained, since it has been acquired by GitLab in 2015 [79]. The fact that Gitolite, which is still maintained, was for a time used by GitLab as an authorization layer, renders it even more relevant, given the GitLab's popularity. One of the reasons for that is that GitLab faced performance issues with an extensive count of repositories and users. [34] This might become an issue for the massive corporations, but

[4]Not impossible – they can be used. They are inconvenient however, because the system requires to be used as a whole, while only utilizing a mere fraction of it. The provided SCM features are not to be used.

since Gitolite performance issues with configuration parsing occurred at over 560 thousand LOC of configuration files and 42 thousand repositories reached by Fedora, it is sufficient for the purpose. [13]

Two examples are distinguished to compare in this section as possible candidates for the authorization tool to use in the project. GitLab and Gitolite are inspected for the purpose closely in the rest of the subsection.

2.3.1.1 GitLab

"GitLab is a single application with features for the whole software development and operations (DevOps) lifecycle." [31] It is an open source project started in 2011 with more than 1900 contributors and used by over 100 thousand organizations as a self hosted Git server with many development supportive features [31].

It offers a rich, well documented GraphQL API (as well as a still maintained RESTful API), which would become beneficial for the application control.

Using GitLab solves the issue of authentication as well, because GitLab comes bundled with an embedded user management service, storing user data in its own database. This is consider a great asset for the purpose.

Regarding the access control, GitLab offers standard control over Git branches via user groups using the *protected branches*[5][35], which is a feature well known amongst similar services. This however remains to be the only level of control it offers within a repository. A file locking feature exists in GitLab, but is only available in GitLab Premium, where it is available since GitLab Premium 8.9 [32].

GitLab is a *single application*, as officially stated. It cannot be used modularly for the thesis' specific purpose.

2.3.1.2 Gitolite

"Gitolite allows you to setup git hosting on a central server, with fine-grained access control and many more powerful features." [15] Gitolite, presumably developed since 2009[6] is an open source authorization layer atop SSH, which controls user access to Git repositories.

The advantage over GitLab for the usage is that it manages solely authorization. Unlike GitLab, using it does not require to inject a large monolithic application only to leave most of its features unused.

Gitolite unlike GitLab offers a truly powerful access control configuration. It features a "wild card" regular-expression defined repository names [12], and a much more advanced feature similar to *protected branches* from GitLab. This offers means of controlling not only branch names, but even tags, paths within the repository and even establish push meta rules, such as changed files count per push. All mentioned using its *vref*[7] and *refex*[8] [14] as seen in the listing 1.

The sample Gitolite configuration in the listing 1 taken from [14] showcases the access settings for the repository *foo*. It grants unrestricted read-write

[5]Protected branches are means of restricting the user access based on Git branches. It usually distinguishes between *read*, *write* and *master* permission, which allows force updates, delete etc.

[6]September 17, 2009 is the first tagged release on GitHub, v0.50

[7]Abbreviation for virtual reference

[8]Neologism formed of *reference* and *regex*

```
repo foo
    RW+                     =   @alldevs

    -   VREF/COUNT/5        =   @juniordevs
    -   VREF/NAME/Makefile  =   @juniordevs
```

Listing 1: Gitolite configuration example

access (`RW+`[9]) to all developers (group called `@alldevs`) and restricts access for the junior developers (restriction using "`-`" symbol) to push more than 5 files and to change the `Makefile`.

This configuration sample demonstrates the power of the fine grain access control Gitolite provides, which is not only superior to GitLab in its expressiveness, but is also stored in simple configuration file in a Git repository available through Gitolite itself.

Apart from that, Gitolite features a group management as does GitLab.

2.3.1.3 Summary

The stated advantages, namely rich concept of virtual references for access control and accessible version controlled permissions configuration outweigh the single, yet considerable drawback, which is need for custom authentication.

Thus Gitolite is preferable over GitLab.

2.3.1.4 How Gitolite works

Before resolving the second issue of user access revealed earlier, which is the approach to unified authentication mechanism on over SSH and WUI alike, details of Gitolite's authentication and authorization details are discussed.

This is important because the system needs to have means of authenticating the user on the web as well as of checking the authorization rules for the repository access, which behaves in the exactly same fashion as the one over SSH.

Note that these are neither an installation instructions, nor an in depth explanation of Gitolite works inside. Just bare essential to understand its basic concepts.

2.3.1.4.1 Install Gitolite is a program typically installed under a new user called *git*. It takes over its home directory and makes necessary changes to it. Git user's home directory holds all repositories, including the administration repository. The administration repository includes access control configuration, as well as the registered public keys for the authentication. Gitolite keeps additional files updated for a successful SSH authentication (discussed later).

The Gitolite installation requires one public key for the initialization. The first key (its user) is granted an access to configuration repository.

[9] the "`+`" symbol means advanced access to e.g. force push branches

2.3.1.4.2 Adding a user Users are added by changing the `gitolite-admin` repository, the mentioned administration repository. It contains the folder with the public keys and the configuration file for the authorization. A new user is added by pushing commits, which add their public key to the `gitolite-admin` repository. This repository on the Gitolite server is the single one to have a `post-update` hook, which creates a record in the `authorized_keys`[10], allowing the new user to authenticate via `git` UNIX user onto the Gitolite server with SSH key-pair authentication.

2.3.1.4.3 Authentication As is obvious from the previous paragraph, Gitolite does not implement any form of authentication. It is relying solely on the SSH layer to perform a secure authentication via the key-pair authentication. Gitolite must provide authentication data by cautiously managing the `authorized_keys` file.

```
1 # gitolite start
2 command="/home/git/bin/gitolite-shell
  ↪ hump",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
  ↪ ssh-rsa AAAAB...VAQ== hump@station1
3 command="/home/git/bin/gitolite-shell
  ↪ dump",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
  ↪ ssh-rsa AAAAB...VAQ== dump@station2
4 # gitolite end
```

Listing 2: Gitolite git user authorized keys file

Gitolite restricts incoming users from full access via SSH configuration. Notice the `authorized_keys` file with running Gitolite with several users in the listing 2. The file `authorized_keys` does not only contain the public keys authorized for access, one per line. Apart from many options irrelevant at this moment, it contains an option `command`. It *"specifies that the command is executed whenever this key is used for authentication. The command supplied by the user (if any) is ignored. (…) This option might be useful to restrict certain public keys to perform just a specific operation. An example might be a key that permits remote backups but nothing else."* [70]

Which means a UNIX user on a machine with running SSHD [71] can control what command is executed for SSH key authenticated users. This can be used to run different a shell, modify the environment or as in this case, to forbid the users to run anything, except a specific program. In this case it is gitolite-shell with provided username argument.

This is important for the further discussion of authentication, since it makes other than the key based authentication impossible to use.

2.3.1.4.4 Authorization Gitolite authorization runs in two steps.

This process is depicted in the diagram 2.3. The activity diagram, though simplifying the details to display the higher-order concepts, contains all the essential components in the communication for the demonstration of the discussed issue. The diagram presumes that SSH authentication succeeds.

[10] Implicitly located in `.ssh/authorized_keys`

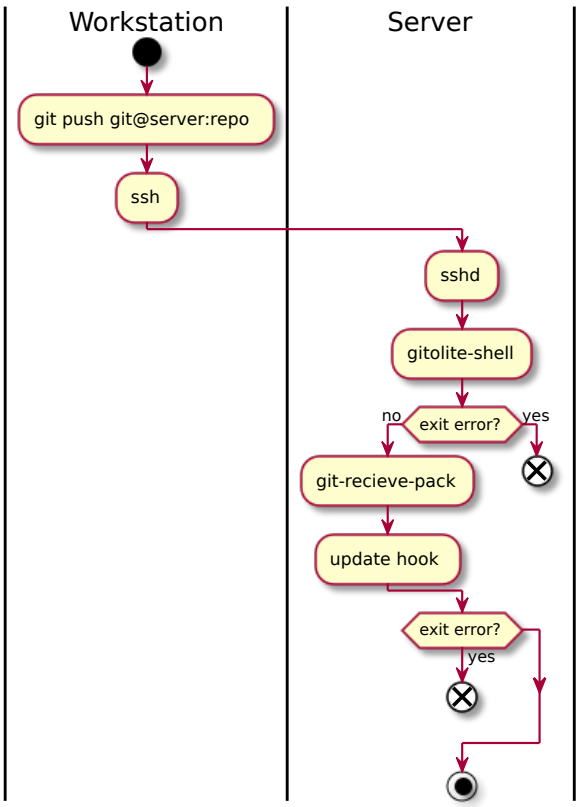

Figure 2.3: Gitolite two step authorization

The first step in the process is to run the `gitolite-shell` with the username and the repository name, supplied via SSH by the remote user. At this point, Gitolite can evaluate (and eventually deny) the access, because it already knows the authenticated username, as well as the repository name and the action (is it a read action, such as `git fetch`, or write, for instance `git push`). If Gitolite does not deny access at this point, Git standard command is invoked, e.g. `git-upload-pack` for cloning or pulling form a repository.

For the read operations the first step is also its final. However that is not true for the writing operations such as `push`. For that, after the `gitolite-shell` command passes, `git-recieve-pack` is invoked instead. This receives and applies the data from the initial push, which eventually triggers an `update` hook. The hook performs additional checks for each updated reference and it may partially or totally abort the update by exiting with an error.

2.3.2 Authentication

Having discussed the authorization layer and its limits, it is apparent that the system requires the SSH key-pair authentication method.

Can the same concept be utilized in the web environment?

Though not technically impossible, it is not definitely a standard approach for the authentication on web. The discussion [74] contains further details on this topic. The main problem of the issue is the access to the local files from JS in the browser. Using a browser extension for that, as suggested in [74] is inappropriate and a *requirement* of user extension an everyday authentication is unnecessarily complicated. Using a key-pair authentication is a non-standard approach to the problem with obvious obstacles. Thus it is not considered for the project.

The solution of authenticating a user and binding it with a SSH key is used by popular similar SCM services.

If the user cannot be authenticated via key-pair, yet it must be guaranteed that the user, no matter the authentication method, is correctly paired with the public key, then the key must be provided by the authenticated user. Binding user and a public key is not possible in a secure way. The solution is then to use the standard ways of authentication on the web and let the user upload their public key via a web application, performing an authorized request under the identity of authenticated user.

This solution is used by giants amongst the SCM services, including GitHub [30], GitLab [33] or BitBucket [7].

2.3.2.1 The standard ways of authentication

Since the SSH key-pair authentication is impractical on the web, conventional way of authentication are briefly discussed.

A common way of authentication on the web is providing a UID (username, email, etc.) and password. The server then retrieves the user by UID from its storage and compares the passwords (the results of hash functions with one of the inputs being the password).

Users are familiar with the method, it is simple and portable – independent of the browser, OS etc.

Implementing this authentication in a secure fashion and keeping it up to date is challenging, and the solution has great re-usability potential. For that reason (but not only as mentioned later) there are services that act as authentication authorities. This allows other applications and services, regardless of the platform, to communicate via HTTPS with the authority and let it authenticate the user instead. This contributes to re-usability of the user's *key*[11] they have – not only improving their comfort but also containment of the personal data in applications specialized for that purpose.

A popular architecture of such service is OAuth 2.0. OAuth 2.0 is primarily an authorization service. The authority manages user's data. An existing application can request authorization for portion of the data. User (after a successful authentication) can grant or reject the authorization of the application for requested portion. When the access is granted, the application can manage the data. Requesting data about the user, the OAuth 2.0 can be used as mere authentication provider.

This is utilized by the OpenID Connect [72], which is a standard based on the OAuth 2.0. It is not a general authorization provider, but an identity provider.

[11] Meaning means of authentication, not a asymmetric cryptography key-pair.

Using an external provider is a viable solution for the system, since it simplifies the authentication process, allows the user to use an existing identity in the system and the application to access user data if convenient.

2.4 Requirements model

Since the system tackles both web application as well as the SSH access to the repositories, it provides two UIs. The former is referred to as WUI and latter as CLI throughout the text.

2.4.1 Functional requirements

Wiki system works with text files formatted in a specific markup syntax. System must provide support for AsciiDoc [5] and Markdown [40] and must allow extensibility for other LMLs using modules. The set of at least two mentioned LMLs (possibly extended by additional ones) is referred to as *supported markup*.

- F-1. **Authentication**

User authenticates via an external authority. After the successful authentication, system provides means for user to upload their public SSH key to unlock CLI. Alternatively the application retrieves the SSH public key from the authority provider.

- F-2. **Authorization**

Each authenticated user accessing the system is denied or allowed access according to the current ACL settings respectively, regardless the used interface. The authentication methods may vary for the WUI and CLI access, but the behavior of the authorization layer is consistent.

- F-3. **Content management**

Authorized users can manage the contents of the wiki. Users can preform general CRUD operations on any content, based on their level of authorization.

The system uses VCS to track changes in the content. Submitted changes create new revisions in a history log, which can be accessed to review individual revisions or restore content from a specific point in time.

The content consists primarily of text files for the convenience of VCS, however it might include binary (e.g. media) files as well.

The system provides convenient editing interface for document files in *supported markup* format. This interface is provided by a specialized editor.

It is not necessary, nor desired to cover all possible user interaction in WUI. The paramount priority is to offer editing interface for the *supported markup* documents.

- F-4. **Content browsing**

Users can browse the published content; this action may or may not require authorization based on the current ACL for WUI. The authorized users can access repository directly via CLI.

The WUI provides an interface for repository file browsing relevant to the current repository revision, as well as a file preview and a list of available revisions with their changes. File preview of the *supported markup* documents interprets the markup and displays a rendered document.

- F-5. **Authorization management**

With the sufficient authorization access, user can edit ACL for the selected repositories. For each registered user and individual repository a read write access can be explicitly allowed or disallowed.

2.4.2 Non-functional requirements

- NF-1. **Storage**

The system stores all the data only in a set of Git [80] repositories on single server machine. That includes the contents of wiki itself as well as e.g. ACL.

The system may use other technologies for other data when convenient for e.g. cache, session management etc.

- NF-2. **UI**

The system provides WUI as well as an SSH direct access (CLI) to the Git repositories.

- NF-3. **Platform**

The system is implemented in JS (the web client interface) and Node.js (server-side).

2.5 Use case model

2.5.1 Actors

The system in general recognizes three types of users. The role hierarchy is illustrated in the diagram 2.4.

1. Anonymous user

The anonymous user can browse or even contribute to a public repository if its authorization policy allows that, but only via WUI.

2. Authenticated user

The authenticated user can browse or even contribute to a public or private repository if its authorization policy allows that, via WUI or CLI.

3. Administrator

The administrator is an authenticated user with write access to a specific repository holding the access control policy.

Figure 2.4: Use case model: Actors

2.5.2 Browsing

The diagram 2.5 displays the use cases for the browsing section.

- UC-1 **Git remote access**

User remotely modifies Git repository via the SSH standard Git interface. The power of the following editing interaction is limitless and not related to the described system, because the changes happen at the user's local workstation.
The interaction of the user is either of the following types:

1. Read operations (`clone`, `fetch`, `pull` etc.)
2. Write operations (`push` and its variations)

If the user is not authenticated and tries to access any repository (including the public ones)[12], operation is not permitted.

- UC-2 **Sign in**

The user can authenticate through external authentication authority. If the authority provides access to the user's public key (e.g. GitHub, GitLab, etc.), on first sign in, the key is paired and CLI becomes available.

- UC-3 **Traverse tree**

User can list files in the currently selected repository. File names are visible and recognized file types are distinguishable in the list.
If list item is a directory, user can select the item to navigate to the directory sublist. In same manner, user can traverse the tree back to root folder.

[12]Necessity of this restriction has been discussed earlier in the section *User access*

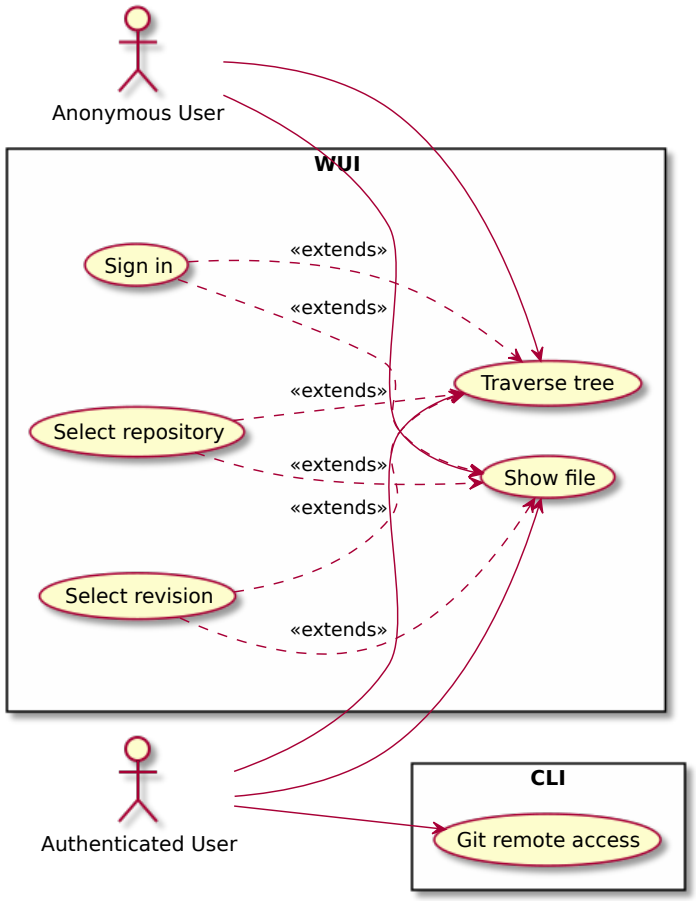

Figure 2.5: Use case model: Browsing

- UC-4 **Show file**

User can display contents of the text file. If the file is a *supported markup* document, the rendered preview is available.
The preview of binary files is not supported.

- UC-5 **Select repository**

User can select a repository. This affects the *Traverse tree* UC.

- UC-6 **Select revision**

User can select a revision for the selected repository. This affects the *Traverse tree* and *Show file* UC.

2.5.3 Content management

The diagram 2.6 displays the use cases for the content management section.

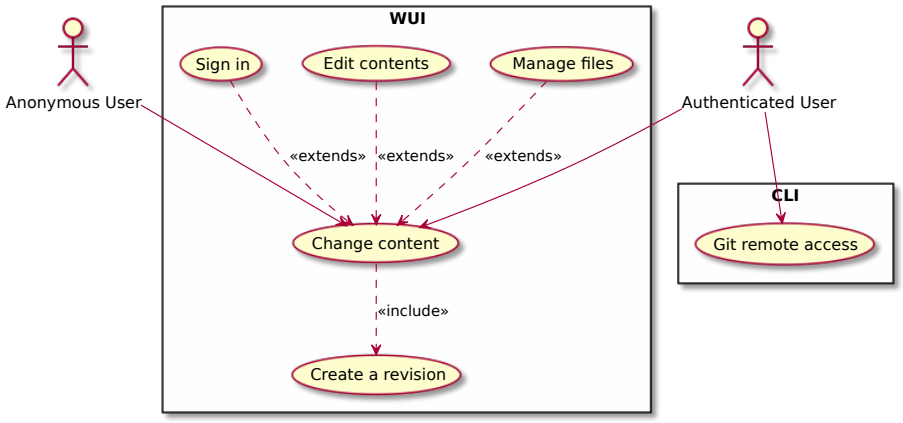

Figure 2.6: Use case model: Content management

- UC-7 **Change content**

If the user is not authenticated or unauthorized to perform edits on the selected repository, use case scenario ends.

User performs any of the *Edit contents* or *Manage files* use cases in any order. User must perform the *Create a revision* use case to complete scenario.

- UC-8 **Edit contents**

The user is presented an interactive editor. If user is editing a *supported markup* document, an editor with specialized features for the given language is provided.

If the file is not a *supported markup* document yet it is a text file, user can edit its contents in simple textarea. Otherwise, the editing is not possible.

The user edits the contents of the file. When they are done, they submit the results.

- UC-9 **Manage files**

The user can rename, edit or delete files in the repository if they are given access.

- UC-10 **Create revision**

The user types a descriptive short message to describe their revision. The user confirms the revision.

2.5.4 Access control

The diagram 2.7 displays the use cases for the user access control section.

- UC-11 **Manage user access permissions**

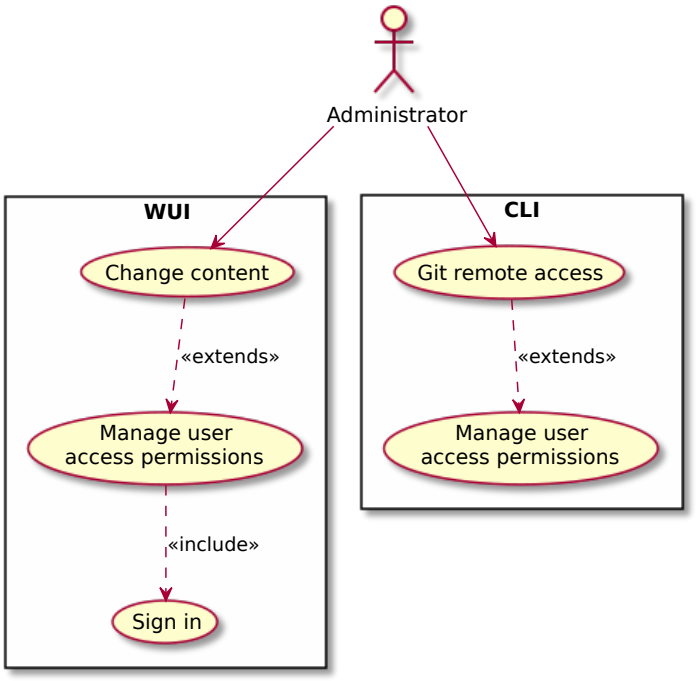

Figure 2.7: Use case model: Access control

The administrator can edit a special repository containing the authorization policy. In this file they can create user groups, grant or revoke access on read and write levels to the existing repositories and their namespaces.

2.6 Use case - functional requirements coverage

The use cases are a mere elaboration and further specification of the functional requirements, with detailed interaction of the user and the system. Therefore by the definition, all use cases shape at least one functional requirement, and each functional requirement is implemented by at least one use case.

The diagram proves 2.8 proves that it is so.

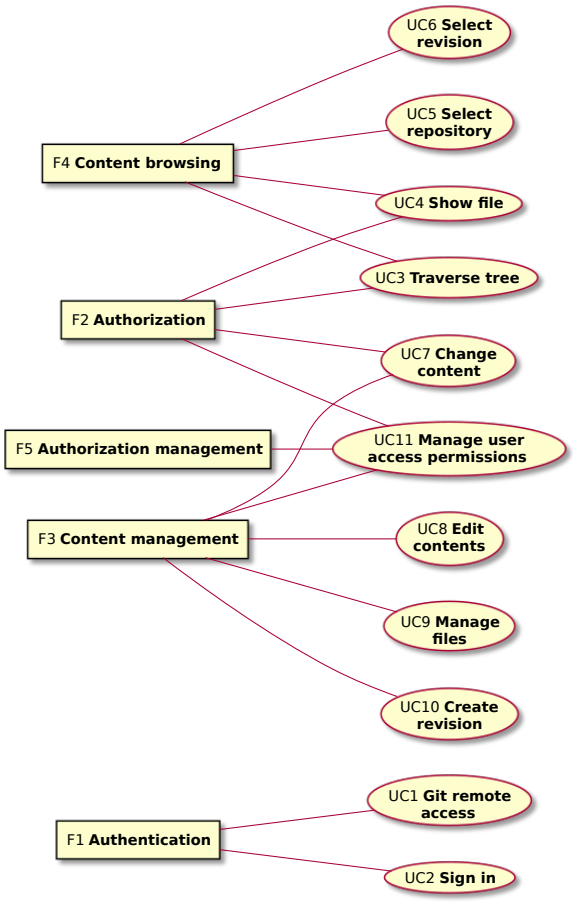

Figure 2.8: Use case - functional requirements coverage

Chapter 3

State-of-the-art

In this chapter, several existing OSS wiki systems are compared. The systems are selected when their design and feature set most strongly resemble the traits discussed in the previous chapter.

Since the requirements are highly restrictive, only the systems using Git VCS for data storage are present, regardless of the satisfaction of the remaining demands.

The remaining requirements are discussed individually with each system. It is described where they excel and where they are inferior. The remaining desired features, apart from Git VCS in the BE, are:

- **User access control**
  - How powerful it is
  - What security scopes are available (protected branches, namespaces etc.)
  - The integration of the configuration into the VCS
- **Direct repository access**
  - The direct access via Git CLI available
  - How the authentication is handled with the web login and the SSH key pairing respectively
- **Document format**
  - Variety of the supported languages
  - Support for Markdown and AsciiDoc
  - Extensibility for the new LMLs
  - Language feature support
- **Branching model**
  - The Git branching model support
  - Parallel version development
- **UI**

– The incorporation of the Git VCS into the UI[13]

The acceptance and quality criteria is defined. Based on the former the several popular systems that either run on the VCS Git or are able to use it as a primary content repository are inspected. The systems Wiki.js, Gollum, Gitit and Ikiwiki, complete the list of all major OSS projects, which are backed, or can be in that matter, by Git.

Some items on the list are not repository hosting services per se, for instance Gollum, which is only a WUI for managing a Git repository. Since the number of the project passing the acceptance criterion is thin as it is, Gollum is included as well, for its UI research value.

The list can be extended if plug-ins or extensions are considered. This way popular platforms such as MediaWiki or DokuWiki could be included. Extensions are excluded however, and only platforms which are designed to work with a VCS repository are considered, for the reason of having a greater research value of the design and the UI specifics regarding the Git incorporation.

3.1 Ikiwiki

Ikiwiki is a software project licensed under GNU GPL version 2 or later [46], written in Perl and was first released in April 2006 by Joey Hess et al. *"Ikiwiki is a wiki compiler. It converts wiki pages into HTML pages suitable for publishing on a website."* [51] What is the most admirable about Ikiwiki is that can use Subversion as well as Git, which implies a form of VCS abstraction is used throughout the system. This can result however, in the lack of features such as access control (because general implementation of this feature is exceptionally complex) or Git specific features such as branching model for parallel development.

Ikiwiki is, as stated, primarily a *wiki compiler*, meaning its main goal is to compile the document pages into the selected format (presentable format from LML).

3.1.1 User access control

For the Git SSH access there is no form of user control when using Ikiwiki with Git. Keys for accessing the repository via SSH must be set manually and are not managed by the application [49].

Access control in the WUI is possible [50] through the `httpauth` plug-in using the CGI configuration. This allows to create a private wiki as a whole, as well as to pinpoint the pages that require authentication (and leaving rest implicitly public).

The subset of pages that require authentication is defined using *PageSpec* [52]. This allows to define the pages as a mere list of their names, but also using advanced functions for matching links to the pages, date creation or pages created by the given user. PageSpec is a surprisingly powerful tool, as shown in the following demonstration. "*For example, to match all pages in*

[13]It is useful, to observe how the UI is affected by the fact that the system runs on the VCS Git. UI presumably shifts slightly from a "document hub" (known from common wiki software, e.g. MediaWiki [87] or DokuWiki [37]) to a broader perspective of a generic repository browser.

a blog that link to the page about music and were written in 2005" [52] use PageSpec displayed in the listing 3. [52]

```
1 blog/* and link(music) and creation_year(2005)
```

Listing 3: Ikiwiki: PageSpec example

The concept of authorization and authentication is intertwined, as officially stated in [53]. Though it is technically possible to somewhat differentiate between the two using PageSpec functions `user(username)`, `ip(address)` etc. and distinguish read, write access for e.g. comments with functions `comment(glob)` and `postcomment(glob)`, the ACL is formed of a single logical expression, resulting in an unmaintainable configuration, when used for several users with a non-trivial access policy.

3.1.2 Direct repository access

The direct access via Git CLI is available, since Git repository can be hosted on the remote machine.

Ikiwiki must be provided the SSH keys for the remote repository access. Serving the repository over SSH is not provided by the software.

3.1.3 Document format

Ikiwiki supports primarily Markdown with extended syntax[14]. However, via plug-ins it features support for HTML, WikiText [89], Textile [4] or reStructuredText [38]. [45]

Extensive document editing is expected to be handled with a direct file access in custom environment. The WUI offers only a simple textarea. Using a RTE in the WUI has been considered and is recorded in the "todo" section [54], however, with an outdated link to the current version 1.6 of the plug-in.

3.1.4 Branching model

It is possible to use the mentioned Markdown extension `directive`, to reference a Git branch [48]. The existing branches [44] are displayed as single documents in the WUI, forming one *bug* or a *todo*. Parallel version maintenance through WUI is unlikely.

3.1.5 UI

On the conceptual level as described in the introduction of the chapter, the UI of the wiki is not affected by the underlaying VCS layer, as seen on the page preview on the image 3.1. There are tools in WUI however, such as the history preview, that take the advantage of the VCS background. Atop of the content of the wiki page with outline by its side, there is a toolbar with links including *Edit*, *History*, etc. and a search form. At the bottom, there are pages linking to this page and last edit meta-data.

[14] Using *wiki links*(`[[WikiLink##foo]]`) and its akin *directives*.

Figure 3.1: Ikiwiki: Page preview

On the edit page there is a standard static preview button. This is depicted on the image 3.2.

3.1.6 Summary

- The PageSpec tool is powerful, but very hard to use for the basic configuration and is maladroit to maintain.
- The SSH authorization is not handled and burdensome to sync with the existing authorization settings.
- The Authentication is blended with the authorization.
- The usage of Git branch is unusual.
- Plain-text editing is provided.

Ikiwiki, being the oldest project in the chapter, suffers from the historical decisions, which might have been relevant at its time. Nevertheless, they

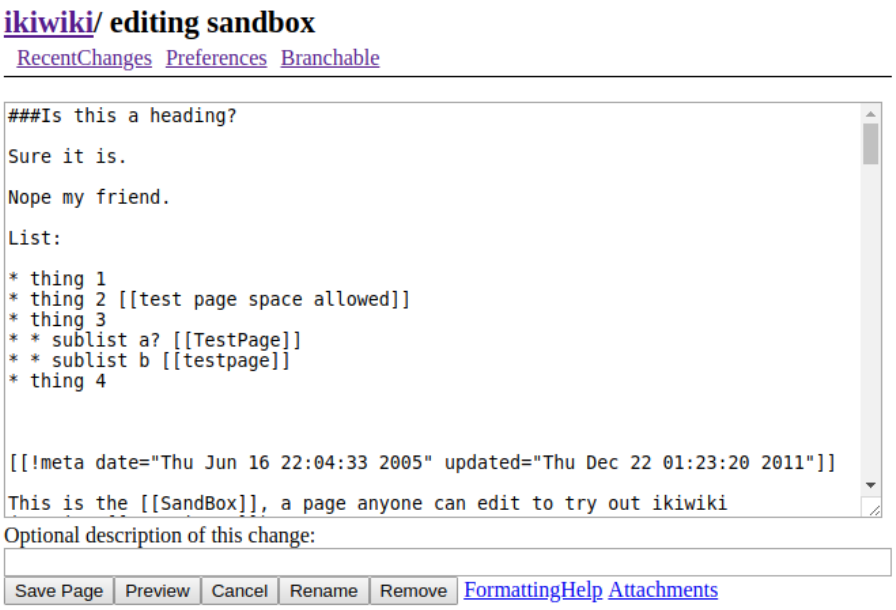

Figure 3.2: Ikiwiki: Page edit

are but burdens now. This includes complex PageSpec implementation and unfortunate fusion of the authentication and the authorization.

3.2 Gitit

Gitit [62], a software by John MacFarlane, is written in Haskell. It made its initial release in November 2008 and brings many features compared to the previous entry. The strongest of which is its flexibility, which can be seen in the VCS abstraction and the document format options as well.

Gitit can use either a Git, Darcs [77] or a Mercurial [63] repository. The rather remarkable document format options are discussed later in the appropriate section.

From the user experience, it is incomparable to Ikiwiki[15] and can be set up in literally few minutes with basic configuration.

3.2.1 User access control

In spite of Gitit having many revolutionary ideas, user access control is its weakest link in the strong chain. Gitit user manual does not mention any form of permission management within the repository.

The only tools it offers for user restriction are:

- setting a global permission level for the anonymous users and

[15] Ikiwiki software and its homepage and documentation wiki are maintained in a single repository [47], with confusing user manual and installation instructions scattered in the wiki.

- requiring a correct answer for the configured access question before registering a new user.

These options are available in the configuration as seen in its sample in the listing 4, which is a snippet from the default Gitit configuration[16].

```
require-authentication: modify
# if 'none', login is never required, and pages can be edited
↪  anonymously.
# if 'modify', login is required to modify the wiki (edit, add, delete
# pages, upload files).
# if 'read', login is required to see any wiki pages.

access-question:
access-question-answers:
# specifies a question that users must answer when they attempt to create
# an account, along with a comma-separated list of acceptable answers.
# This can be used to institute a rudimentary password for signing up as
# a user on the wiki, or as an alternative to reCAPTCHA.
# Example:
# access-question:  What is the code given to you by Ms. X?
# access-question-answers:  RED DOG, red dog
```

Listing 4: Gitit: Configuration sample

Gitit provides permission control only in the global scope. It does however let the user select, whether to use implicit user file storage for the new users, or to allow GitHub OAuth 2.0 authentication. With this option, setting the client's credentials in configuration file is required.

3.2.2 Direct repository access

The direct repository access is on the same level of support as was the case with Ikiwiki. The system does not provide the external access to the Git repository. If user desires to permit such access, it is solely their responsibility.

3.2.3 Document format

This is where Gitit truly excels beyond its rivals. Though lacking the specialized tools for the selected LMLs, it offers

1. a large variety of supported languages and
2. an export option of the page into a rich set of document formats, including other LMLs (Markdown, MediaWiki, AsciiDoc, etc.), typesetting formats (e.g. LaTeX, ConTeXt), office document formats, DocBook, sideshow formats and much more, using Pandoc [61].

The documents are implicitly written in the Pandoc's extended version of Markdown. In a YFM (document preamble), meta-data including format can be set, as seen in the listing 5. The supported formats include reStructuredText, LaTeX, HTML, DocBook and Org [19] markup. [62]

[16] Which is a result of `gitit --print-default-config`

```
---
format: latex+lhs
categories: haskell math
toc: no
title: Haskell and
  Category Theory
...

\section{Why Category Theory?}
```

Listing 5: Gitit: Page preamble example

3.2.4 Branching model

Gitit WUI does not support branching model. It faces the restrains of the premise supporting Darcs VCS which does not have [76] a branch support. Git repository itself is not limited to use a single branch, but the application only assumes linear development.

3.2.5 UI

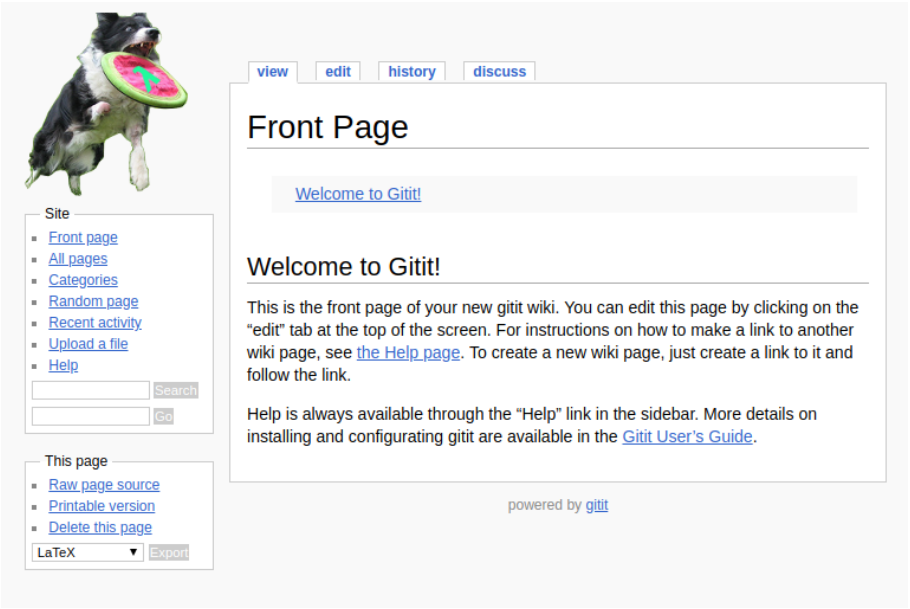

Figure 3.3: Gitit: Page preview

The UI, though appearing more modern, is fairly similar to the previous entry Ikiwiki, as seen on the page detail in the image 3.3. There are improvements, that include navigation tools: for instance *All pages* index and *Categories*[17]. Minor, yet welcoming change, is that links atop the page contents are

[17]Categories can be assigned to individual pages in YFM as seen in the listing 5.

tabular. The links are semantically more akin to tab widgets, than navigation links. This design option, known from other popular wiki software, e.g. [87] is an appealing change, that eliminates user confusion with the navigation and current state of the view.

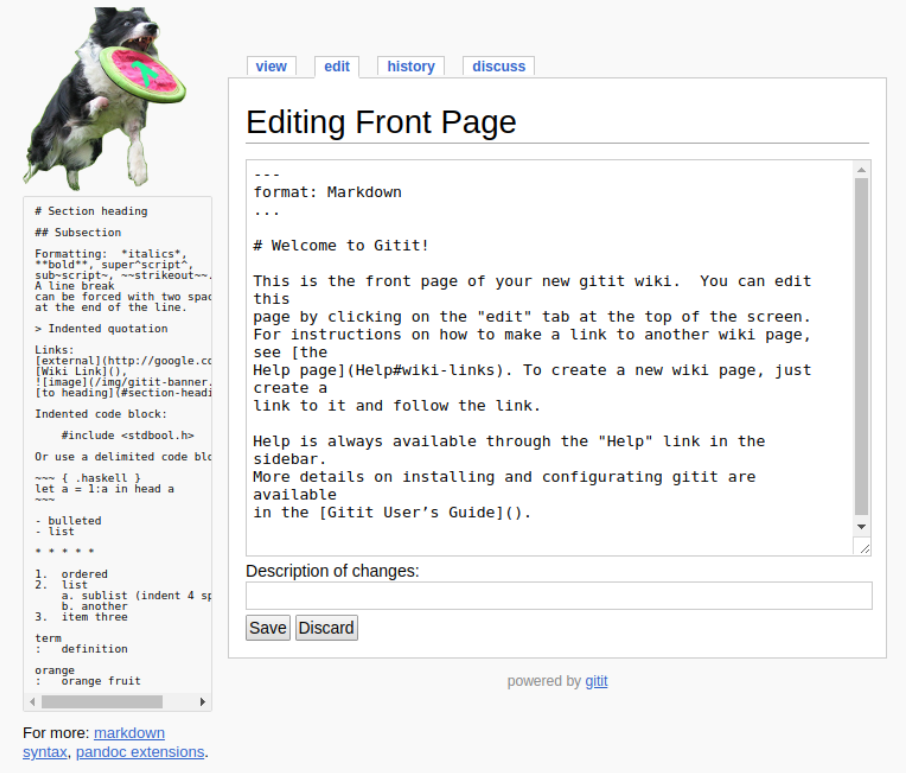

Figure 3.4: Gitit: Page edit

As expected of the format variety, no tool is used for the document editing, but plain textarea. The form UI and the component layout is in this case almost identical to the previous entry, as seen in the image 3.4, though featuring a Markdown cheat-sheet.

3.2.6 Summary

- Gitit can be backed by a Git, Mercurial or Darcs repository.
- System lacks any form of authorization mechanism and leaves only an option to select private or public wiki.
- A wide variety of the supported markup is provided.
- Notable export options are offered via the Pandoc conversion tool.
- Interesting usage of YFM is used for meta-data, which is independent of the LML from Pandoc's perspective.

Gitit profits from its generic approach of using a Pandoc meta document format, allowing it to store meta-data in an unified manner as well as providing

extensive export abilities.

3.3 Gollum

Gollum is an OSS written in Ruby developed since 2009 that powers [91] GitHub Wikis. The Gollum, though being a wiki system, is rather different from the other systems mentioned in this chapter. While its differences make it destined to fail in many criteria, it proves useful to review the system, notably from the UI perspective, especially given the fact that it has been used in GitHub Wikis.

While Gollum does have wrappers or extensions that do provide, e.g. user authentication and permission control, for instance [17], the bare library is *just a WUI* for the repository management with the focus on LMLs.

3.3.1 User access control

Gollum does not handle any form of user control; neither authentication, nor authorization – every visitor can perform any operation on the repository through the WUI.

3.3.2 Direct repository access

Direct repository access is possible, though it is handled by the user outside of Gollum.

3.3.3 Document format

Gollum by default supports Markdown and RDoc [83]. This can be even extended to AsciiDoc, Creole [57], MediaWiki, Org [19], Pod [86], reStructuredText and Textile. With the extensions this is the largest set of supported markup languages.

3.3.4 Branching model

Gollum can actually work with various branches. It can be launched on any branch using the program argument from CLI `gollum --ref=dev`. The default option is `master`.

3.3.5 UI

Gollum is visually minimalistic, yet the user widgets and layout remains still very same, as appereant from the image 3.5. There are navigational options hidden under the *All* and the *Files* options in the toolbar above the page.

There is an exceptionally impressive UI for page editing. Not only there are many document formats supported, as discussed, but Gollum also provides a toolbar (seen in figure 3.6) customized for the given format. Apart from that, it features a static as well as a live preview[18].

[18]That is available when started with option `--live-preview`

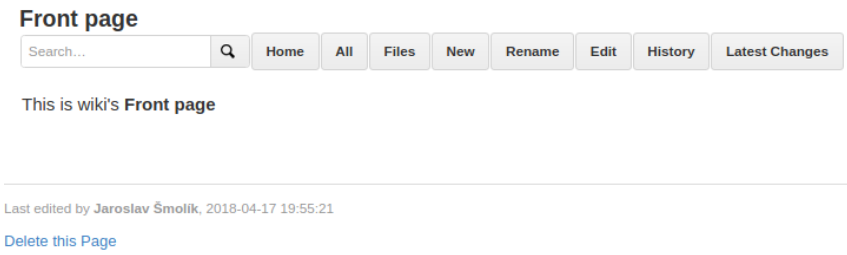

Figure 3.5: Gollum: Page preview

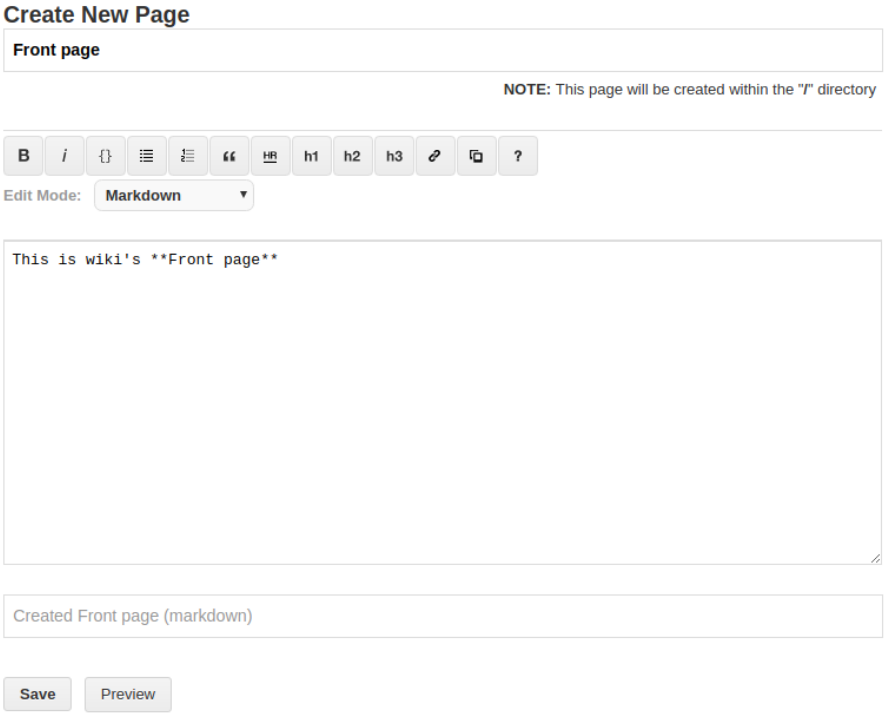

Figure 3.6: Gollum: Page edit

3.3.6 Summary

- Gollum is just a web interface for the repository editing and lacks any form of permission control or even authentication.
- The software supports many LMLs.
- Gollum provides superior document editing UI with impressive options including toolbars for users unfamiliar with the syntax and a live-preview.
- The direct repository access is not managed by the application.
- Gollum features inconvenient branch support, requiring multiple instances to run in order to manage parallel versions.

Gollum steps out of the line amongst considered systems, being "a mere" WUI for repository management. It has an exceptional LML support with an admirable UI.

3.4 Wiki.js

Wiki.js is a modern capable wiki software powered by Node.js, Markdown and Git [26], developed by Nicolas Giard et al. With the initial release in September 2016, Wiki.js is the youngest software.

Being designed with the specific technologies in mind, the specialized features are be expected from the project. As far as the non-functional requirements and technological restrains reach, Wiki.js, being a Node.js application, backed by Git only[19] and favoring modern popular LML Markdown, is by far the closest to the thesis' project.

After the installation user is prompted to run the configuration wizard[20], where user can set the name of the wiki, configure MongoDB[21], default permissions (e.g. is wiki public to anonymous users by default) and a remote Git repository with its SSH authentication data.

3.4.1 User access control

Wiki.js is the first project in the chapter to offer a strong ACL mechanism. Within the WUI, administrator can set permissions for individual users or managed groups via settings. The permission rule, added to a user or a group consists of the following settings:

- Permission – either *Read only* or *Read and write*
- Path
  - *Path starts with* or *Path match exactly*
  - Path string
- Access – either *Allow* or *Deny*

[19] The abstraction of used VCS can potentially be a threat, as seen with Gitit not supporting branches

[20] Configuration can also be set in the `config.yml` configuration file.

[21] Which is used for user data, not wiki's contents.

Though the described ACL interface is not as complex, nor as powerful as seen among others[22], it is most fitting for the created scenario – a relatively easy way to restrict access within repository in a name-space manner.

Though not apparent from the documentation, after an examination of the MongoDB database data of running Wiki.js, it is obvious that they are actually stored in the database, rather than in files. This means that ACL is not implicitly version-controlled nor available for the direct access provided by Wiki.js.

3.4.2 Direct repository access

Contents are stored in a Git repository. This can either be a local repository, or a repository mirrored to its remote over SSH or basic authentication.

Repository is by its nature accessible directly from the remote and Wiki.js handles [28] the synchronization on its own. Git repository is clean, formed solely of the Markdown documents. All meta-data are detached from the repository and saved in the MongoDB.

Wiki.js however does not handle direct repository access[23] and it is in the hands of the administrator again.

3.4.3 Document format

Wiki.js only supports Markdown as the sole document format. This can be seen as a disadvantage in form of a user restriction as well as an advantage – the potential for Wiki.js to peruse perfection in the tools specialized for Markdown.

The same way Gitit used the Pandoc format's YFM for meta-data, Wiki.js uses a similar trick. Since Markdown does not support any meta-data syntax, Wiki.js uses Markdown's comments[24] for meta-data. This syntax can be used to define page title for instance as seen in the listing 6.

```
1 <!-- TITLE: Home -->
2 <!-- SUBTITLE: A quick summary of Home -->
3
4 # Header
```

Listing 6: Wiki.js: Markdown meta comments

3.4.4 Branching model

The branching model is not supported. Wiki.js can change the remote branch for synchronization, but this requires the change in the configuration.

This is a similar approach to the one taken by Gollum, which effectively requires a restart to lock onto a different branch.

[22] Meaning it does not provide as complex expressions as seen in Ikiwiki's PageSpec, for instance.

[23] in sense of a hosting service

[24] At least its "most supported" syntax of the comments. Since there is no consensual cannon specification of the Markdown grammar, it is defined by its various implementations. Their somewhat unexpected behavior can be compared in [60].

This feature can be actually used for the parallel development, when running multiple Wiki.js instances and mirroring to a single repository, but onto different branches. This strategy is useful for developing few, not-related wikis, mirroring into a single repository. This approach however, is not useful for managing several versions of the same repository of a single project, the way it is described in the business process model.

3.4.5 UI

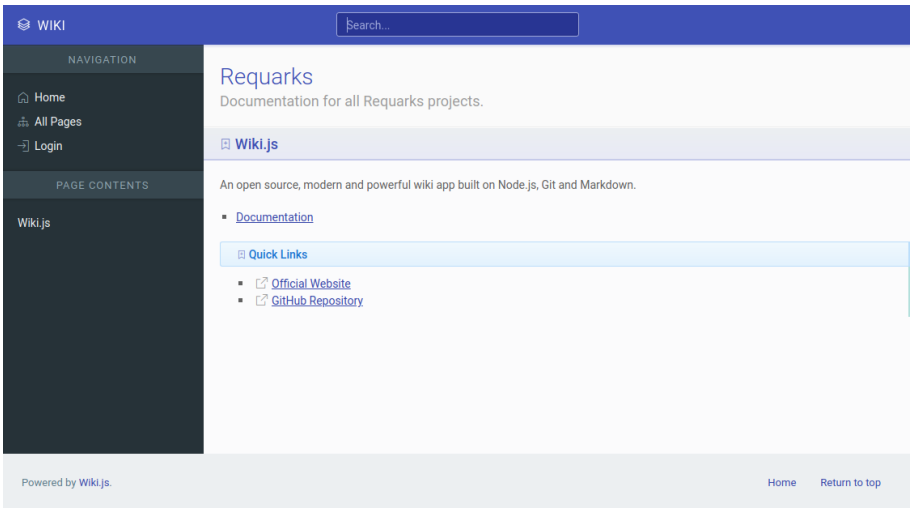

Figure 3.7: Wiki.js: Page preview

Wiki.js offers the UI (image 3.7) which has the same widgets as seen before, though visually distinguished from other project. This is plausibly an effect of the fact that Wiki.js is the youngest software. Apart from the website navigation there is a document TOC navigation placed under it in the left sidebar.

Wiki.js offers custom Markdown editor with features focused on the language, as seen in figure 3.8. The editor features a hybrid live preview of the formatted source code (using proportional sizes and font styles and colors for the formatted markup), as well as a toolbar for the users, who are unfamiliar with Markdown, who can use editor as a RTE.

3.4.6 Summary

- A user friendly, relatively powerful permission control is provided.
- Branch support is available, though inconvenient for the scenario.
- SSH authorization is not handled by the application and it is burdensome to synchronize with the existing authorization settings, since FE and BE API decoupling is in progress a scheduled [27] for the version `2.0`.
- Only Markdown LML is supported.
- Hybrid advanced RTE is provided.

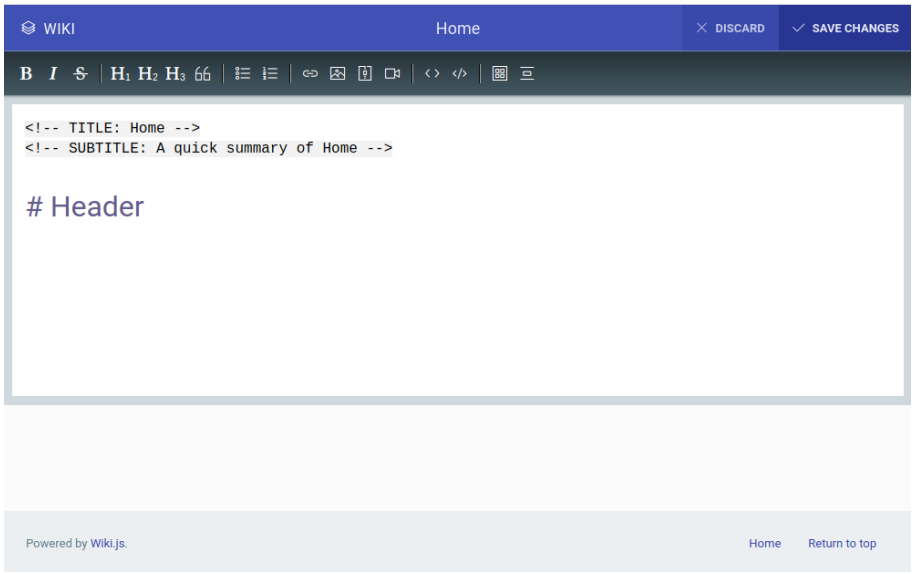

Figure 3.8: Wiki.js: Page edit

- Git repository mirroring is available.

Wiki.js provides convenient user permission configuration, modern UI and specialized Markdown editor.

3.5 Summary

The selected wiki systems are discussed in this chapter with regard to established criteria. None of the systems are ready to substitute the software product of the thesis, as concluded in the defects (regarding the given criteria) in each summary.

The major issue with each solution is failing the requirement *F-2 Authorization*, which states that the authorization policy is consistent, no matter the used interface[25] The only solution that has an in repository permission control is Wiki.js, and as stated in its section, the permission rules are stored in a database with no guarantee of consistent API provided by Wiki.js at the moment. Interpreting the Wiki.js' permission configuration is not feasible. Having said that, even if it were possible to obtain the ACL configuration from Wiki.js, its power is far less sophisticated than the one provided by the Gitolite configuration. Having Wiki.js permissions compiled into the Gitolite configuration would degenerate the ACL expressiveness to the level of Wiki.js.

Apart from that, there are few other important, though less fundamental issues.

Firstly is the breach of the *F-3 Content management* which demands support for Markdown and AsciiDoc as well as modular extensibility. Some systems, such as Wiki.js, are bound to specific formats, so extensibility is not in

[25]One user is bound by the same set of rules whether they use WUI or CLI.

question. The only system that by brief glance plausibly satisfies the requirement is Gollum, which alas failed at the following issue.

The issue majority of the systems fail to comply is *UC-6 Select a revision*. Many systems ignore the repository's branching model and others offered an impractical implementation of it.

From the point of the UI that would distinguish Git backed wiki systems from others, the main aspect is the navigation. While generally wiki uses namespaces as a hierarchical structure of pages, said systems usually favor directory perspective[26] and provide a *file browser* widget for traversing the repository directories. This is clearly seen in Gollum for instance.

VCS revision (Git commit) is in all systems considered as an atomic[27] change. User cannot not create a commit via the WUI that would change the contents of the two different files for instance. Rename, delete, create commit messages are created automatically, without the user's knowledge, while revisions changing file contents, prompt the user to describe incorporated changes manually.

In the next chapter, it is explained how are the stated issues resolved.

[26]This is the cause of the fact that the Git repositories usually contain at least some portion of source codes, or other files that are not pages per se.

[27]in sense of files

Chapter 4

Design

In this chapter a conceptual design of the system is presented along with the solutions for the issues raised in the summary of the previous chapter.

4.1 Design foundations

There are two conceptually different approaches to the implementation of the unified authorization layer for both WUI and CLI:

1. allow remote repositories, while losing the control over them, or
2. keep the repositories managed exclusively by the system, allowing for firm permission control potential.

4.1.1 Remote repository, limited permission control

This is the approach more or less taken by all the reviewed systems in the previous chapter. The premise is to allow the users to work with their existing repositories and remain their remote locations. The implementation is akin to how Wiki.js tackles the issue – the application works with the local mirror of the repository, which is kept synchronized with the remote, thanks to provided access data, such as repository link and security configuration (either the HTTP Basic or the SSH key-pair authentication).

The implementation possess the following attributes:

- Easy setup and installation (no need to configure the SSH server)
- Central public application instance can be used by the community, users can drop in or out at their convenience
- Possibly larger base of users would be addressed due to the previous attribute

4.1.2 Local repository, potentially extensive permission control

This approach is used by the larger SCM services like GitLab. The system is not just a web application but also a Git hosting service. This gives the application ultimate control over the repositories. The custom Git hosting

service allows to create a complex permission control layer for remote access (or utilize an existing software for the purpose).

- More challenging for unexperienced users to setup the hosting service
- Central application instance is not feasible
  - users are not likely to give away their repositories to an unknown provider
  - decentralized ACL administration is difficult to tackle, even with a sophisticated tool as Gitolite
- Thus, few users actually get to use the system

4.1.3 Conclusion

The former design likely reaches to more users, while the latter provides powerful control over the repositories, allowing for a solid authorization layer.

The concept of *repository providers* is described in the following section, which provides a solution to overcome the obstacle of having both: the remote repositories and the firm control over the local ones in parallel.

4.2 Repository providers

The key to bringing the benefits of both, radically different architectural approaches, is to create a solid abstraction layer for the repository within the application, as well as for the means of retrieving and publishing them – the repository providers.

4.2.1 Repository provider's API

Repository provider is a module that can:

1. **Accept authentication data** – Accept an API key, username etc. to use within the module, when it is accessing private repositories.
2. **List available repositories** – Return a list of available repositories it has access to, based on the authentication information.
3. **Obtain a repository** – Clone a repository from an existing remote into a temporary space, likely FS[28], or update its references if it already exists. This action either returns a repository abstraction or fails due to a network error, an unauthorized access etc.
4. **Create a revision and publish** – The provider can block the action and result in an error, or pass the revision to the repository abstraction for commit. The abstraction applies the commit and publishes selected branch to its upstream on the remote repository. Provider can disallow this action based on its implementation or response of the remote.

[28] Also remote FS or any other abstraction, which can be accessed to perform changes in the Git working directory

4.2.2 The advantages of using repository providers

4.2.2.1 Unified approach

Using a tool, such as Gitolite, it is possible to tamper with the original repositories, which are eventually located on server's FS. While tempting in the short run, it is very short-sighted.

Utilizing direct FS access needlessly over-complicates the core logic, forking it into working with an authentic original, and a mirror of a remote. This effectively branches the behavior of the publishing process for instance, and closes the gate to concurrency control.

Therefore, for design purposes it is far mote suitable to treat local repositories as remote, handled by an unified provider interface.

4.2.2.2 Repository abstraction

Treating the local repositories as remotes proves beneficial. Not relying on the local repositories and always aiming for the mirror of a remote, creates a generic abstraction, ready to be used in both scenarios.

4.2.2.3 Application access control

The repository provider module has control over every action of the repository: access, edit and publish. It can, deny access depending on its own logic. This is required for the local provider. The paramount objective is to offer unified authorization mechanism over SSH and WUI.

Instead of accessing the repositories directly, with the concept of the providers, a local provider can be designed. It fetches the repositories from (and publishes to) the local machine over SSH.

Simulating the the user's remote control over SSH would be an ideal solution, if it was possible[29]. Instead, the behavior is simulated through the application logic. Gitolite provides a CLI with a sufficient API to tell the user, which Gitolite user has access to which repository.

Even though the Gitolite configuration is stored in text files, it is better to use the Gitolite CLI tool to parse the permissions. The configuration can be rather complicated, as seen the listing 1. Thus, it is better to use an existing parser, delivered by the same system.

A security bug in application results in exposing the repository data to the users, bypassing the Gitolite security system[30]. As mentioned earlier, the only way to go through the Gitolite-standard SSH access (and not actually bypass it) requires the users' private keys, acquiring which is naturally not possible, as it would compromise the core security principles of asymmetric cryptography.

Letting the repository providers perform additional authorization check, creates a mechanism flexible enough to create even as complex provider as the local provider, which communicates with Gitolite.

The local provider interactions are shown in detail in the diagram 4.1.

[29]This requires user's SSH private key.

[30]The application requires the master SSH key-pair to clone any repository.

4.2.3 Designed providers

To showcase the flexibility of the system, as described in the two design pillars in the *design foundations*, one additional provider is designed. GitHub repository provider is chosen for its popularity amongst the OSS community.

This brings two main advantages for Gitwiki:

- GitHub is still easily the most popular SCM service in comparison to GitLab or BitBucket. Addressing the GitHub's user base is more efficient than other services' providers.
- GitHub, being an OAuth 2.0 provider, is used in the system as an authentication authority. Moreover the user's public keys can be loaded into Gitolite from GitHub through its API.

4.2.3.1 Local repository provider

The diagram 4.1 displays an interaction of the local provider with Gitolite for actions:

1. *List repositories*,
2. *Get a repository* and
3. *Create a commit*.

The provider communicates with the application's wrapper for Gitolite CLI. Gitolite CLI can be requested to list repositories and answer, whether a user can access given repository.

When anonymous user asks for repositories, Gitolite lists for repositories accessed by user `@all`. This is a special user[31] placeholder for *all users*. This results in a query for the authorization configuration: *What repositories can be accessed by all users?*

The design of the local provider solves the issue of unified authorization raised in summary of the previous chapter, by intertwining the application logic with Gitolite's authorization layer through the Gitolite CLI.

4.2.3.2 GitHub repository provider

In contrast to the local provider, a GitHub repository provider is designed. The repository provider API, allows for the providers to use any form of asynchronous communication with their resources. In this case, GitHub's RESTful API is used.

The diagram 4.2 shows the GitHub provider performing the same tasks as required from the local provider. For simplicity it is presumed that the user Alice has valid access and no errors occur.

4.3 Authentication

In the previous section it is implied that GitHub OAuth 2.0 is used for the authentication. That however does not bind the project to GitHub's specific needs.

[31] or a repository placeholder, based on the given context

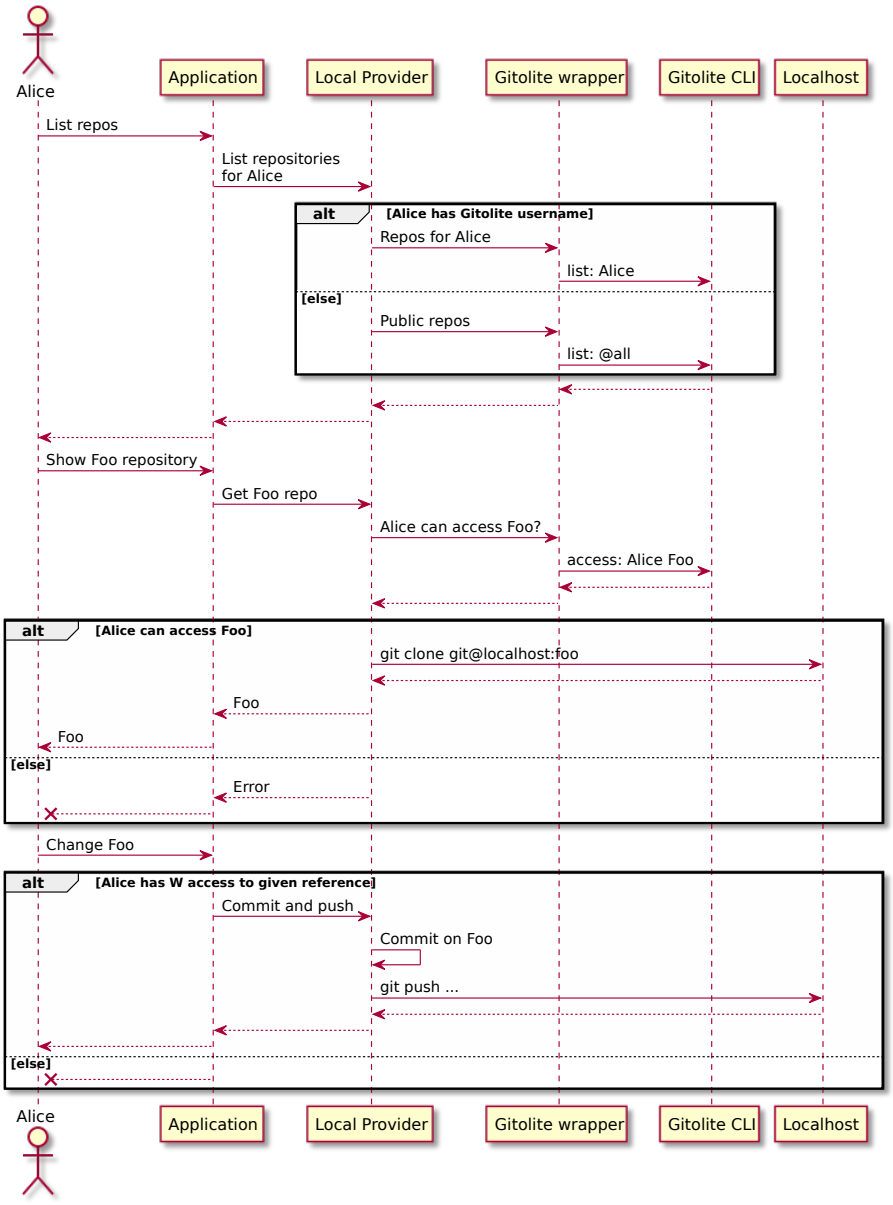

Figure 4.1: Design: Local provider interactions

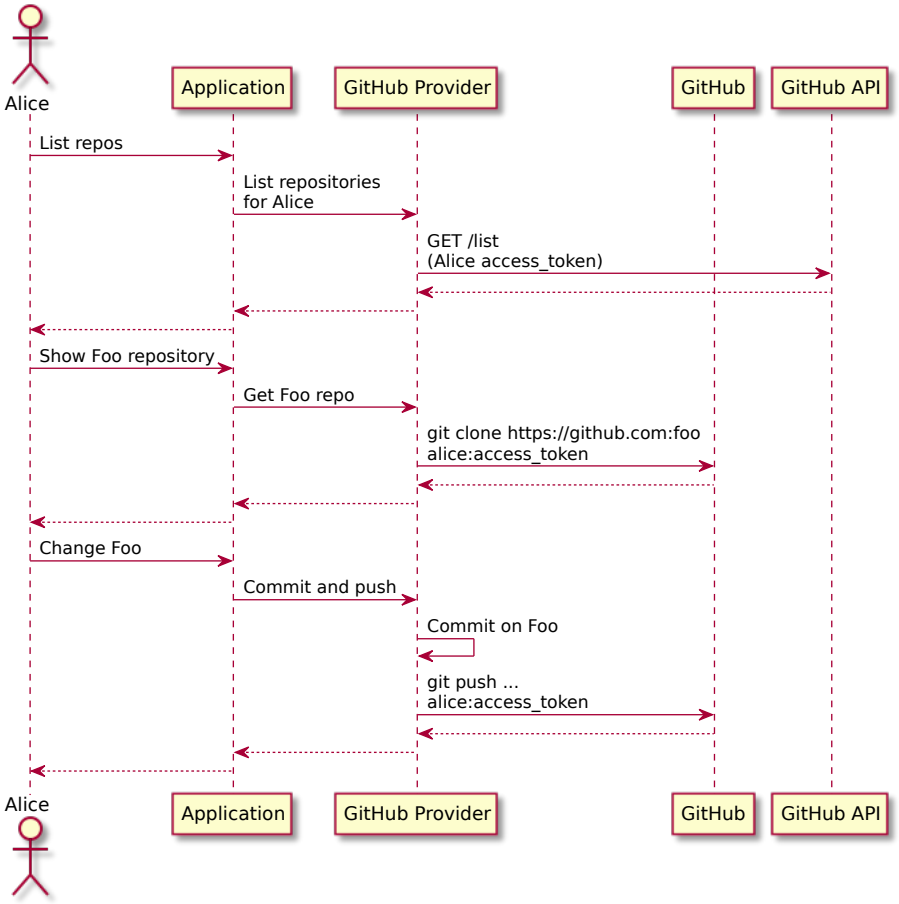

Figure 4.2: Design: GitHub provider interactions

The GitHub authentication is used as a form of authentication, but it does not set course of the project in any major way. The same way, any form of external authority can be used as an authentication provider. Any SCM service[32] is more suitable for the role of the authentication provider, because they provide the user's SSH public keys.

In the same manner Google, Yahoo or any other service with OAuth 2.0 or OpenID Connect protocol support could be used instead. GitHub is a reasonable choice in the scope of OSS projects hosting statistics.

The sequence diagram 4.3 displays the authentication process of using an external authority. Provided that the authority has the user's public key, setting up the new user for using Gitolite can be automated within the first login. The diagram follows the basics of OAuth 2.0. After the user selects the authority, e.g. GitHub, a request is sent to application's BE, which makes an authorization request for the given authority, doing which it provides required scopes of the authorization. The scope names are arbitrary and specific to

[32]Including GitLab, BitBucket and similar

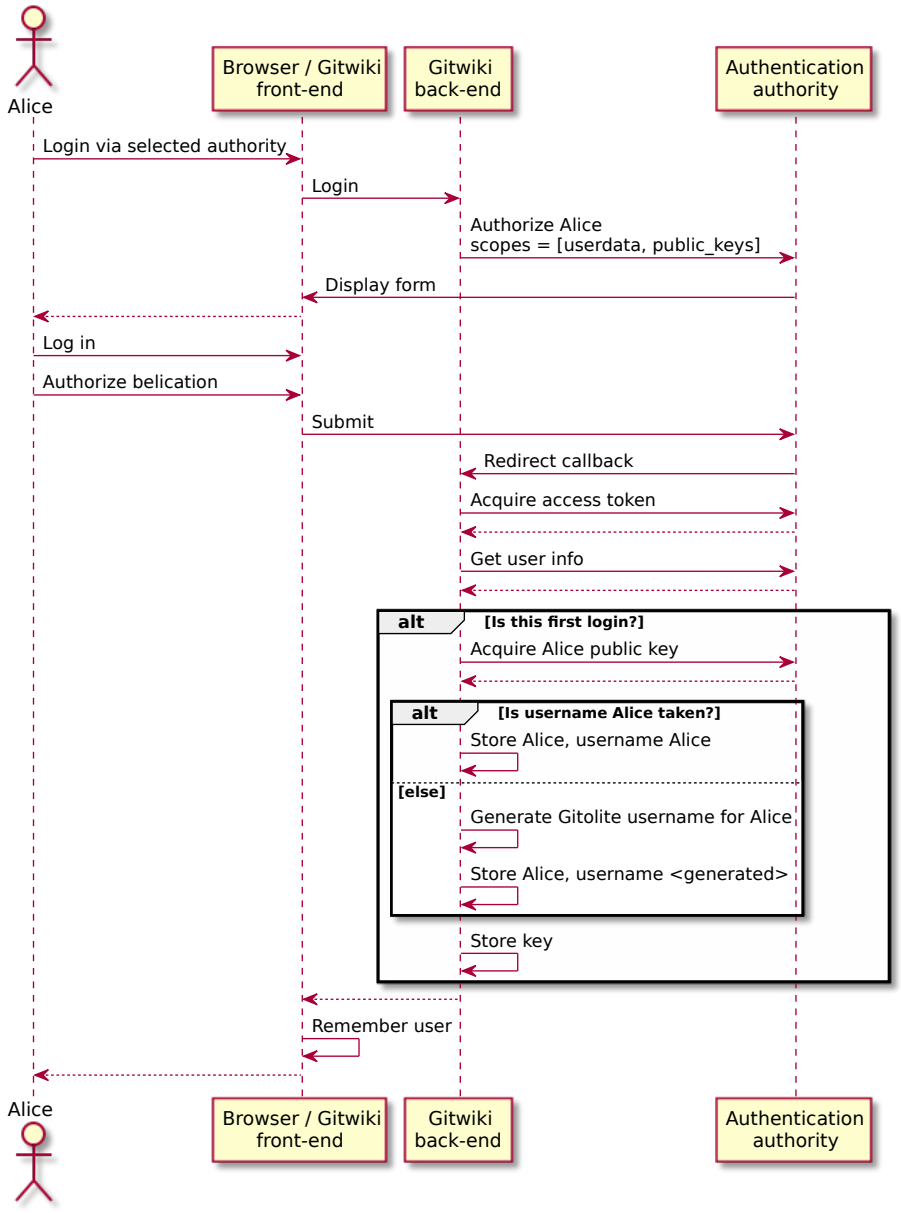

Figure 4.3: Design: Authentication via external provider

each provider, thus configured in the application – the names of the scopes in the diagram are purely illustrative. The authority redirects the user to a page with a HTML form, where the user logs in and authorizes Gitwiki for the given scopes. After submitting the form back to the authority, it is processed and returned to the defined callback URI, the BE endpoint, whence the application finishes the authentication process, acquiring the access token. The token can be used to fetch the user data. Provided that the authority has means of obtaining the user's SSH keys, they are retrieved. The user info is stored with either username from the user data, or a generated one, in the case of lack of such data or conflict with the existing user. The user is stored with his Gitolite username and its keys as well.

4.4 Technologies and tools

No web application nowadays is written from a scratch. There are various tools to fasten up the development process and make it easier.

In this section, it is stated what tools and libraries are used for the project. There are several issues that are usually solved by an existing library. It is discussed in this chapter and not in the implementation, because in some cases, the used tools affect the architecture or the components of the designed system.

Since there are several system components, that can be designed using existing libraries and they have far lesser overall effect on the system[33], the details of the libraries and its alternatives are abbreviated.

4.4.1 FE framework

FE of the application is no more a trivial jQuery script with a a few user interactions in the browser, such as it was fairly popular several years ago. More and more tasks are expected to be performed inside the browser for better UX and more dynamic effect.

This also calls for a solution for building complex FE applications. There are few popular solutions for FE frameworks, such as Angular 2, React and Vue or Ember.

React is chosen, being fairly the most popular among its alternatives and fast and easy to use, thanks to the JSX syntax and its virtual DOM.

4.4.2 SSR

Many websites rely too much on the FE technologies, resulting in a monolithic so-called *single-page applications*. When majority of the logic happens in the browser and the server's HTML response contains (more or less) only the `<script>` tag, several issues arise. The hardest problems face the clients that cannot interpret JS. This might seem banal, because generally speaking all the users browse the web with a browser which JS engine. The issue becomes more realistic, when non-user agents are considered, such as search engine spiders for instance.

[33] Compared to the Git SSH authorization layer e.g., which required more structured analysis

This issue is tacked by several libraries, from which Next.js [95] with native support for React is selected.

4.4.3 FE State container

Facebook introduced [22] the Flux architecture in 2014 to solve complexity of MVC's nontransparent dependencies for complex systems with many models and views via a linear unidirectional data flow. The main idea behind Flux is to linearize uncontrolled flow between models and views through a single component.

This component is called *dispatcher*. It consumes *actions* and updates the *store* as a reaction. The *store* defines the *view's* state (*view* renders data from *store*), and can pass new *actions* to the dispatcher. This flow is displayed on the diagram 4.4.

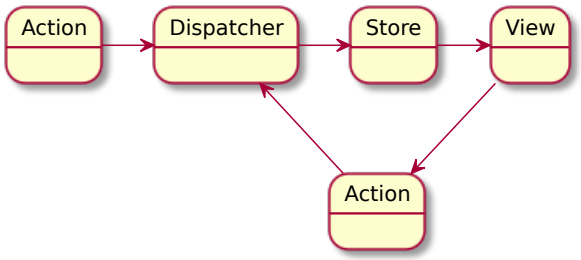

Figure 4.4: Design: Flux architecture

Flux is a general architecture concept and though having its Facebook's implementation of the core modules, there are many other existing options.

One of those is Redux [1], which is used in the project. There is also a Redux-Saga [20] extension for asynchronous action consumption (synchronous consumptions are handled by so-called *reducers*[34]) and works well with Next.js.

4.4.4 Git library

While there are also many libraries for working with Git repositories for Node.js, NodeGit [9] is selected. NodeGit is a binding to a popular C library libgit2 [69], an implementation of core Git methods. NodeGit is used for repository abstraction and manipulation.

4.4.5 Node.js web application framework

Express.js is used for the BE application as routing service and HTTP server with middle-ware management.

[34]Modules that produce new state based on the previous state with regard to dispatched action

4.5 Architecture

4.5.1 Top level architecture structure

A high-abstraction design of the application's architecture is displayed in the diagram 4.5.

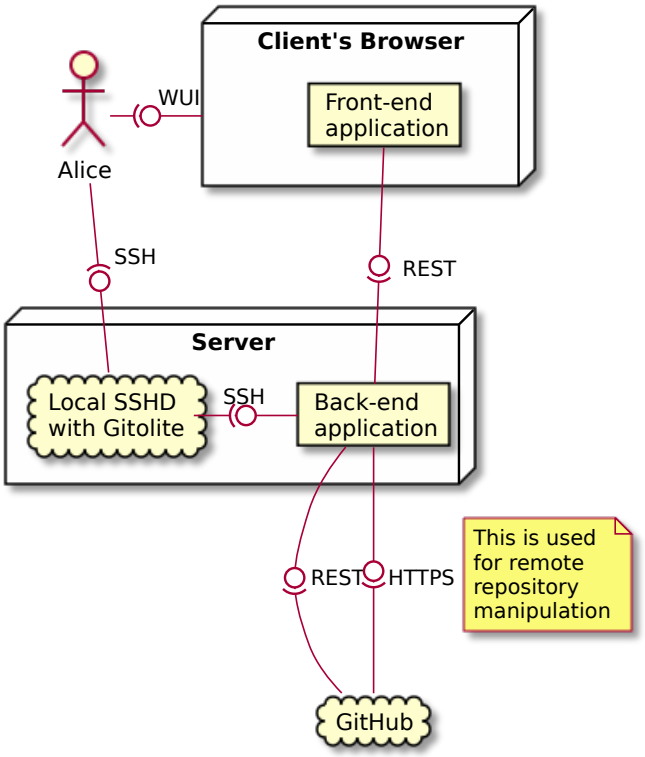

Figure 4.5: Design: Architecture of the application

The application is decoupled into two main components, which are the FE JS application running in the client's browser and Node.js BE application.

FE application is served as a response in user agent's initial request and then communicates with the server's BE application via RESTful API as user navigates throughout the application or performs any actions that require data.

The server machine provides the role of a repository hosting service via the SSH protocol using the SSHD running on the machine. BE application communicates with the self-hosted repository service via loop-back with configured SSH keys, as described earlier in this chapter.

The application communicates with GitHub using RESTful API and HTTPS. As seen in the diagram, server's repositories are exposed via SSH protocol.

4.5.2 BE structure

The essential BE structure is showed in the diagram 4.6.

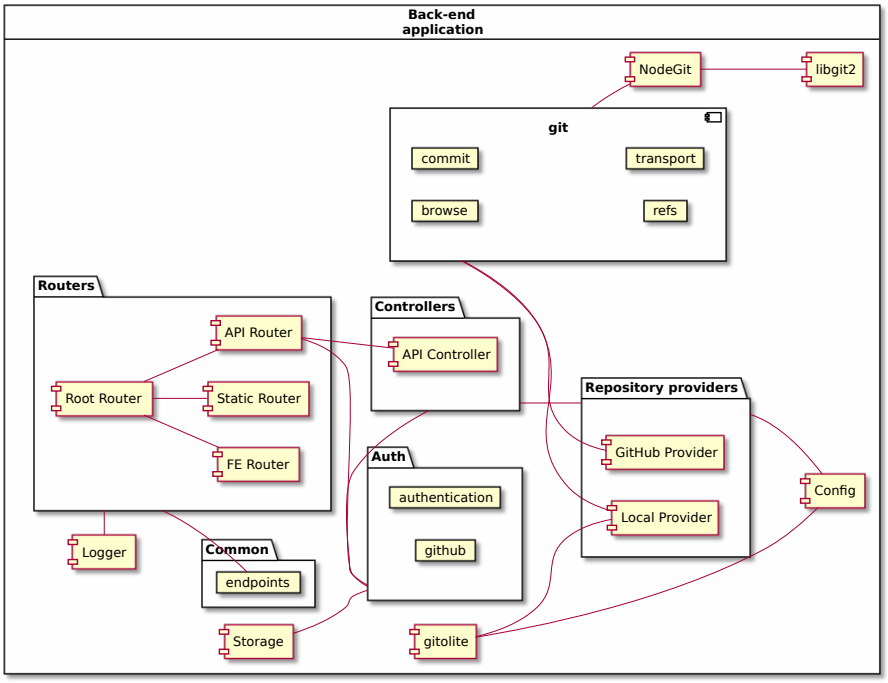

Figure 4.6: Design: Architecture of the BE application

4.5.2.1 Routers and controllers

Routers bind routes defined in the endpoints to the controllers and transform known errors to HTTP codes and appropriate responses.

There is no controller for the FE requests, nor for static files. FE requests are passed onto Next.js handlers, which responds to basic error codes and static router is as well handled by an existing service – Express.js static router. No further decoupling by a controller layer is necessary for the two.

The controllers always return a Promise[35] and prepare response after gathering the required data from the application. The data is acquired from the repository providers or from the `Auth` package, when gathering user information.

4.5.2.2 Repository providers

All the providers decorate and setup the parameters for the `git` module. This includes setting the authentication callbacks, repository URLs etc. Local provider communicates with the `gitolite` module, which is an interface for the Gitolite CLI wrapper.

Providers generally access shared configuration through `Config` module's API, which is an interface for easy access to required parts of the hierarchical configuration file.

[35] for the convenience of uniform handling by routers

4.5.2.3 Git module

The Git module is a set of high-abstraction methods over Git repositories to suit Gitwiki needs. It uses third party library NodeGit, which provides libgit2 bindings to Node.js. The two libraries are shown in the diagram, but only for the sake of communication. Neither is part of the implementation.

4.6 RESTful API

In this section the crucial parts of the RESTful API of the BE application are documented.

For the type definition in this chapter, the Flow type alias [24] syntax is used. Since some object types are reused in the data, common *entity* types are described in the listing 7.

The overall overview of the API is presented in the table 4.1. The following subsections describe the individual API endpoints. Only parts of the API that are interesting from the perspective of either data or design are covered in the chapter. The major logical issue is the design of a commit creation in REST architecture, which is discussed later in the section.

```
type Entry = {
  name: string, // "README.md"
  path: string, // "path/to/README.md"
  isDirectory: bool, // false
  sha: string, // "673d6dcc58fdd8ef6530177ef90bb2c5d1748c34"
};
type Blob = Entry & {
  content: string, // "# Hello world\n\ntodo"
}
type Reference = {
  ref: string, // "refs/remotes/origin/master"
  group: string, // "remotes"
  name: string, // "master"
  compoundName: string ,// "origin/master"
}
type Change = {
  path: string, // "path/to/README.md"
  content: string, // "new content"
  remove?: bool,
}
type Repository = {
  name: string, // "gitwiki"
  description: string, // "\gls{Git} based wiki system"
  provider: string, // "github"
}
```

Listing 7: Entity types definitions

4.6.1 Tree

4.6.1.1 `GET /api/v1/repos/{provider}/{name}/tree/{ref}/{path}`

Table 4.1: RESTful API overview

Section	Method	Url template
Tree	GET	/api/v1/repos/provider/name/tree/ref/path
Tree	PATCH	/api/v1/repos/provider/name/tree/ref
Tree	GET	/api/v1/repos/provider/name/refs
Repo	GET	/api/v1/repos
Auth	GET	/api/v1/user
Auth	GET	/api/v1/auth/github
Auth	POST	/api/v1/auth/github/personal-access-token
Auth	GET	/api/v1/auth/github/cb

```
{
  tree: Array<Entry>, // Current tree
  blob?: Blob, // Current blob entry with content
}
```

Listing 8: REST: GET Tree response

Table 4.2: REST: GET Tree response codes

Response code	When
200	success
401	user not authenticated and accessing private data
403	user authenticated but unauthorized
404	provider, repository, ref or path not found

The tree defined by the relative path `{path}` from repository `{name}` from provider `{provider}` at Git reference `{ref}` is returned. The response JSON structure is defined in the listing 8. Its response codes are displayed in the table 4.2

4.6.1.2 `PATCH /api/v1/repos/{provider}/{name}/tree/{ref}/{path}`

```
{
  changes: Array<Change>, // Changes to commit
  message: string, // commit message
}
```

Listing 9: REST: PATCH Tree request body

A Git commit with supplied changes and given commit message on the repository defined by the request's URL as in the previous example is created. Use user's credentials as *commiter* and *author* from `Authorization` header.

The `PATCH`[36] is unusual with regard to RESTful APIs. However, there is a very special scenario calling for special solution, which is the `PATCH` method. If

[36]The `PATCH` method does not have as strictly defined semantics by the conventions in the RESTful APIs unlike methods `POST`, `PUT` or `DELETE`, which could be used instead.

a `POST` method would be invoked on the tree instead, its semantics would be *creating* a tree and *not* updating (following the RESTful conventions). Updating (and creating alike) could be achieved by using a `PUT` method. When using `PUT` or `POST` however, it is expected to provide the resource, in this case the tree (not *changes*). Following the conventions, the `DELETE` method should be used only for the rare case of only removing a single file or subtree. This concept is very confusing and allows only one change per commit.

Using `POST` on a resource `/commits`[37] allows to perform multiple changes in one commit. However the commit data are not available in the FE. What is available, is a sequence of changes describing the commit. Thus using `POST` is again not an ideal solution with regard to RESTful API. Instead a `PATCH` is used.

The JSON structure of the request body is defined in the listing 9. Its response codes are in the table 4.2 (ditto).

4.6.1.3 `GET /api/v1/repos/{provider}/{name}/refs`

```
1 Array<Reference>
```

Listing 10: REST: GET Refs response

Retrieve the available refs from the repository defined by the URL. The JSON structure of the response is defined in the listing 10. Its response codes are listed in the table 4.2 (ditto).

4.6.2 Repository

The repository section contains an endpoint for listing the available repositories. It has standard return codes as discussed in the previous endpoints (excluding 404) and returns JSON of `Array<Repository>` as defined in the listing 7.

4.6.3 Auth

Auth section provides endpoints for fetching user data, the authentication via GitHub and uploading GitHub personal access token, which is used for cloning GitHub repositories via HTTPS.

4.7 UI

In this section, the designed WUI of the application is presented in a form of wireframes.

The most significant part of the UI is doubtlessly the editor. For this reason, it has been designed separately by a team of students as a MI-NUR term project. The project has been completed and successfully submitted and is available [101] online. The project tackles not only design of the UI, but also its testing, using a heuristic analysis and conducted usability testing. It

[37] Hypothetical idea, the resource does not exist in the API

contains evaluation of the testing methods and fixes the encountered errors. Therefore, since the project completed all the UI development process of the editor, it is not considered in the design of the remaining parts of the system. In the wireframes it is replaced by a simple box.

I have worked with a team on the UI design of the editor. My colleagues contributed only to the project [101] referred to, but not included in the main content of the thesis. The wireframes from the project are available in appendix. I, the author of the thesis, am also the author of all the following UI design regarding materials in this section. Members of the team have not contributed to the implementation of the editor nor the system.

4.7.1 Repository index

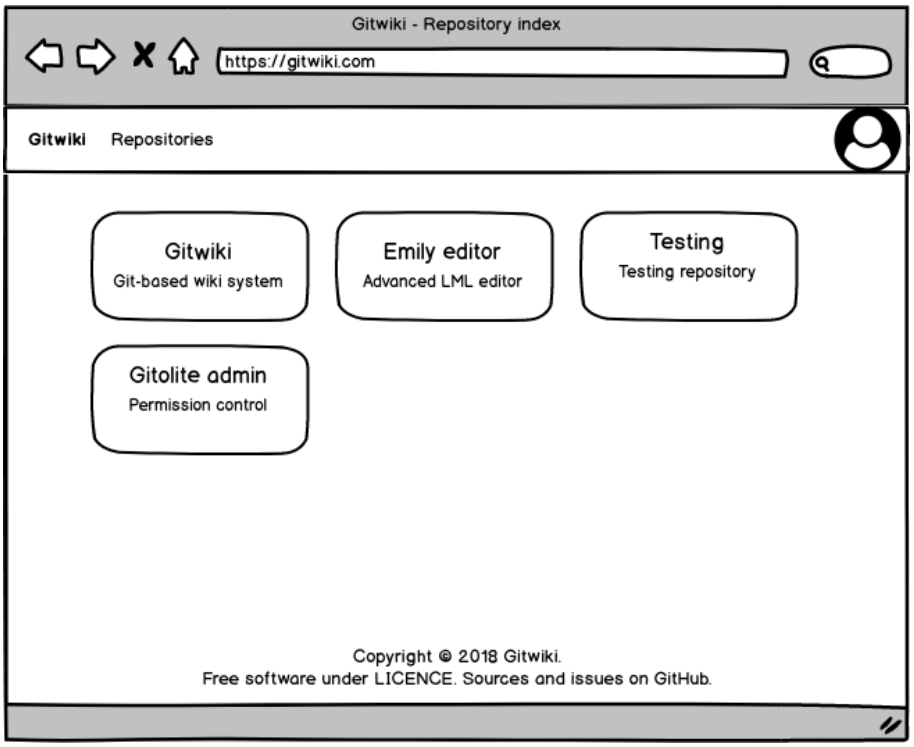

Figure 4.7: Wireframe: Repository index

The basic layout with the navigation bar is displayed on the wireframe in the image 4.7. The navigation bar holds few menu items, a logo with a link to the homepage at the left and a user widget on the far right, as is conventional. The footer holds basic license and ownership or contact info as expected. This components form the layout and are present in all the screens.

This wireframe covers the *UC-5 Select repository.*

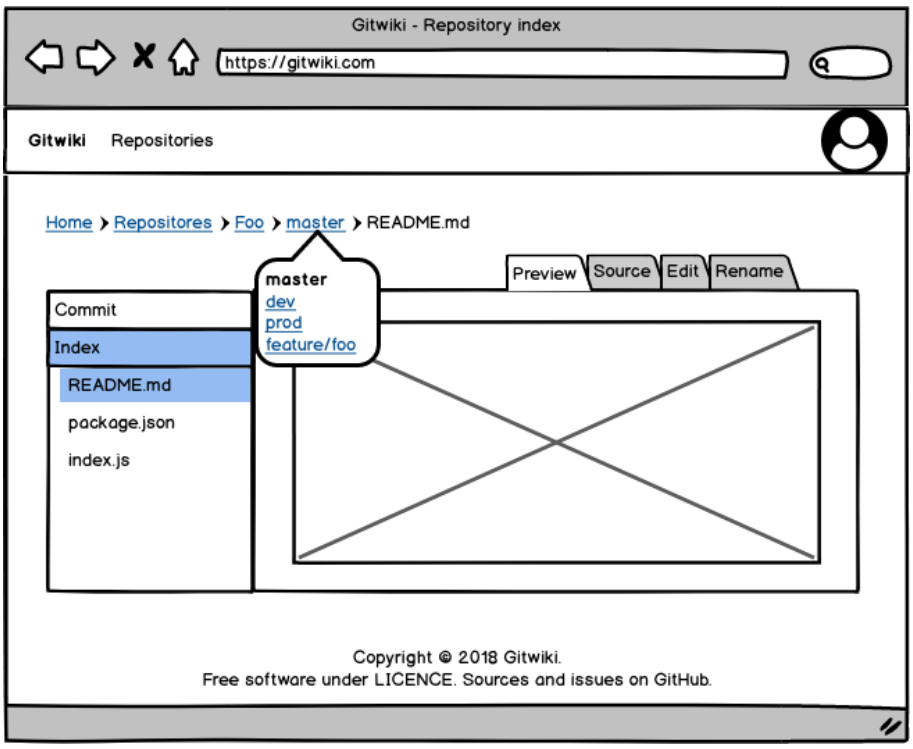

Figure 4.8: Wireframe: File preview

4.7.2 File preview

The wireframe on the image 4.8, presents the file preview.

The breadcrumbs menu contains not only the repository and the fragmented path navigation to the file, but more importantly, a fragment of the menu that displays the Git reference. That is an interactive widget user can use to swap branches. This solves another issue[38] from the previous chapter's summary, which is branching model with a parallel development support.

There is a sidebar menu and the main content. The side-menu holds the link to the commit screen and a list of the files in the current directory. The main content is tabbed and provides an interface for switching between the following views of the current file:

- Preview (if available)
- Source code view
- Editing form
- Rename form

This tab contents always fill the box placeholder. This wireframe covers the following use cases: *UC-4 Show file*, *UC-7 Change content*, *UC-8 Edit contents*, *UC-9 Manage files*, *UC-11 Manage user access permissions*

[38] From the point of UI, the logic has been discussed in the RESTful API design section.

4.7.3 Repository index

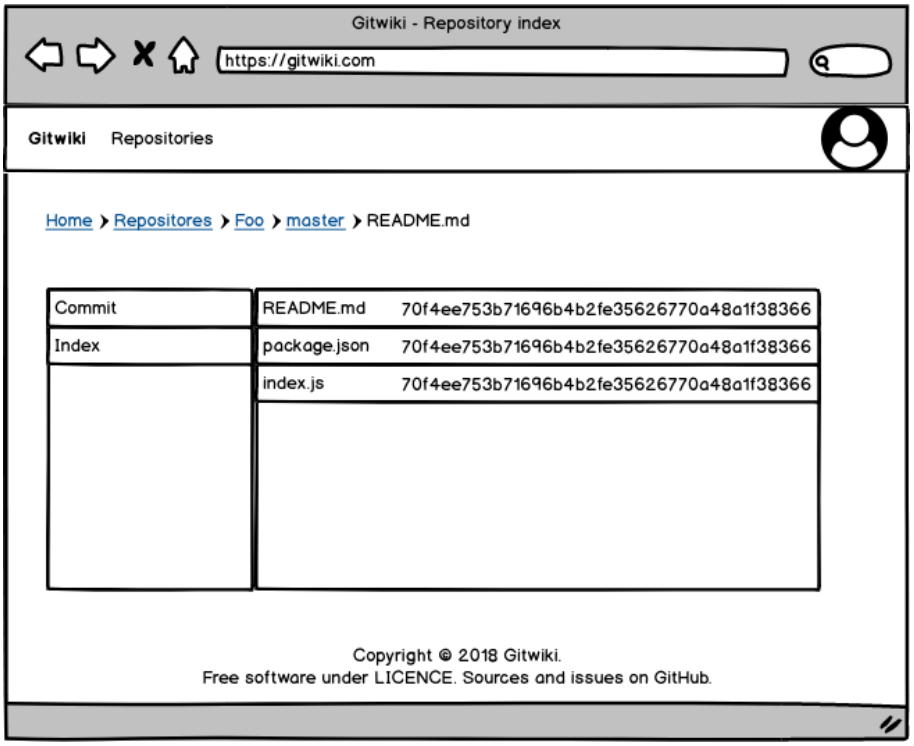

Figure 4.9: Wireframe: Repository tree

In the image 4.9, there is a wireframe to the file index in the repository. This is an expanded version of the side-menu from the previous wireframe with additional information, such as SHA hash to better utilize the room given in the main content section. This wireframe covers the *UC-3 Traverse tree* and *UC-6 Select revision*.

4.7.4 Commit modal

A modal window is used for the commit screen, as seen in the image 4.10. This way it is easily accessible, no matter the current page. The window shows a brief summary of the accumulated changes, input for the commit message and the *Cancel*, *Commit* and *Discard changes* buttons. *UC-10 Create revision* is covered by this wireframe.

4.8 Front-End

The FE application is built of the following composites:

- Next.js pages[39]

[39]Next.js pages are technically React components as well, only decorated with required

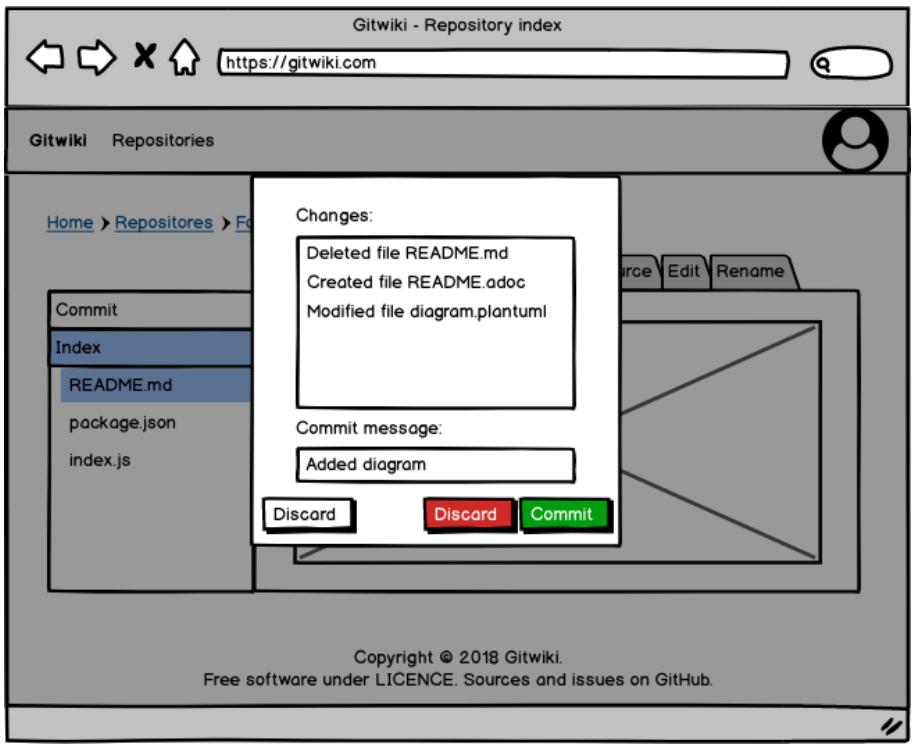

Figure 4.10: Wireframe: Commit screen

- React components
- Redux actions, reducers and sagas
- other utility modules

All crucial components and modules from categories above are on the diagram 4.11.

4.8.1 Pages

The pages are React components representing individual web pages (HTTP responses) served by the server for a user's request. Simply, they are responses for all non-API requests.

When used correctly, Next.js returns the first requested page (pre-rendered on the server) and all the following navigation happens inside the FE application using Fetch API managed seamlessly by Next.js.

- `Index` – Homepage of the application
- `Repo, Index` – List of available repositories. This page delegates the rendering of the repository entries itself on the component `Index`.

features. Semantically however, they are distinct from simple components in a Next.js application.

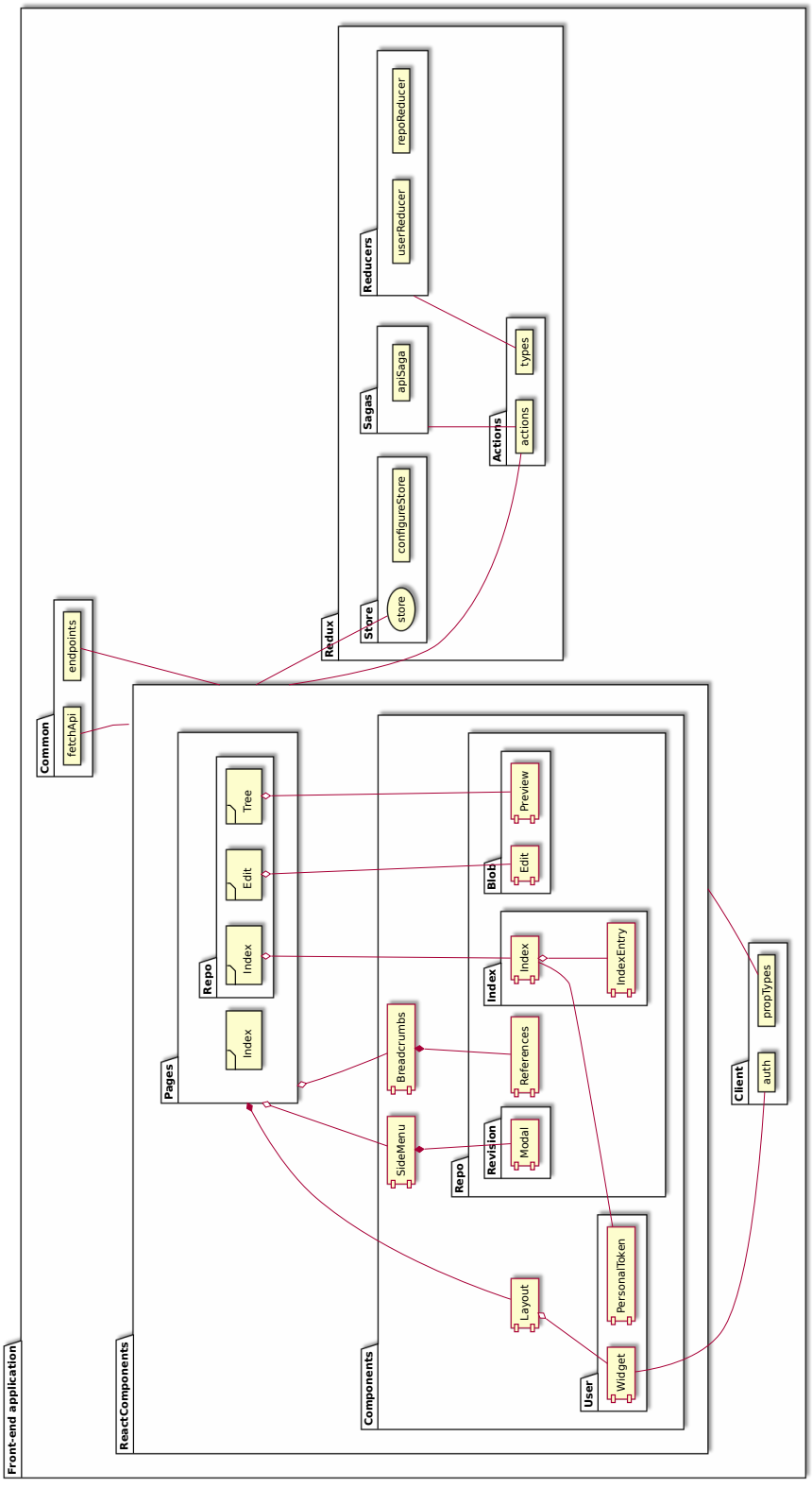

Figure 4.11: Design: Front-end application

- `Repo, Edit` – Edit page for the selected repository, reference and path. The editing logic is delegated to the `Edit` component.
- `Repo, Tree` – Preview of the tree or file on the path, for the given repository and reference. If the path defines a file (not a folder), its rendering is passed to the `Preview` component.

The pages are wrapped in the `Layout` component with shared components (user widget, navigation bar, footer, etc.). Pages can pass `Breadcrumbs` or `SideMenu` component to the `Layout`.

4.8.2 Components

Components represent composite or complex UI elements wrapped in a usable unit within the React application.

4.8.2.1 Layout

- `Layout` – HOC component wrapping page contents with reusable layout widgets. It uses `Widget` to display logged in user data in main navigation bar.
- `SideMenu` – Context-relevant secondary navigation. This includes button triggering commit modal, thus using the related component.
- `Breadcrumbs` – Breadcrumbs menu uses `References` component.

4.8.2.2 User

- `User, Widget` – Displays the login button or the logged-in user's data.
- `User, PersonalToken` – Form control allowing the user to publish GitHub personal access token to access their repositories.

4.8.2.3 Repo

- `Repo, References` – Breadcrumb fragment widget, which loads available references and takes care of the navigation using Git references.
- `Repo, Revision, Modal` – Modal window with the current changes and a form for the commit submission.
- `Repo, Index, Index` – List of available repositories. Single repository record is delegated to the `IndexEntry` component.
- `Repo, Index, IndexEntry` – Single repository entry component.
- `Repo, Blob, Edit` – Form component for editing a given file.
- `Repo, Blob, Preview` – File preview component, allowing to switch between the source code preview and the rendered document in case of supported markup file.

4.8.3 React components

The components and pages from the diagram 4.11 are all React components, as stated before. The both groups are tellingly wrapped in a folder in the diagram, hence they share some features.

All *react components* share PropTypes[40] definitions from the `client` package.

Pages and components alike generally access `endpoints` for navigation (generating links)[41] and `fetchApi`, which is an interface for making requests to the server. All Pages and some components are connected to the Redux store, from which they can access data and dispatch actions.

4.8.4 Client

This package holds modules usable by the client code and not the server. There are PropTypes definitions, which is convenient, for they are usually shared or composited. Apart from that there is a small module for persisting user session in the *local storage*, for keeping the session in the FE application.

4.8.5 Common

`Common` module is for code usable by both FE and BE. `Endpoints` module provides definitions of the routes, which are accessed by BE router as well as used when creating navigation links within the FE application. `FetchApi` is a proxy service for making requests to the server's API. This module in the `Common`, not the `Client` package for the following reason: Having the proxy capable of handling requests made by server to itself is a smart decision, which eventually allows to define initialization of the pages uniformly, regardless the fact that it is rendered in the browser or on the server due to SSR.

4.8.6 Redux

The Redux package includes codes for defining and managing the FE application state.

The `Action` package defines action types constants[42] as well as the action creators, which create action objects from the given data[43].

The `Reducers` are functions which take the dispatched action and transform the existing state into a new one, based on the action's type. Reducers operate in a synchronous fashion.

The `Sagas` are tools for creating side effects. They can react to the triggered actions, but they cannot create a new state. They can only dispatch new actions. This is useful for asynchronous operations.

`Store` is implemented by the Redux library and only its configuration is required.

[40] PropTypes [25] are a type-checking mechanism for validating React *props* passed onto React components.

[41] Endpoints are located in the `common` package, allowing the server and the `client` to access the identical set of path definitions.

[42] This is how action is identified. Every action in Redux has a *type* property with an appropriate string identifier.

[43] Usually just setting the mentioned type property.

4.9 Emily editor

The *Emily editor*[44] is a web based document editor component for LMLs.

The UI design of the editor took place in the [101]. This required not only the wireframe modeling, but also the definition of the editor's functionality.

4.9.1 Editor's features

The result of the feature brainstorming[45] within the team in the early stage of the project is as follows:

4.9.1.1 Feature bag A

- Go to line
- Auto-complete
- Search
- Syntax highlight
- Line numbers
- Text wrapping
- Section folding
- Distraction free mode

4.9.1.2 Feature bag B

- Full-screen mode
- Two-column preview (source code and rendered preview)
- Command palette
- Status bar

4.9.1.3 Feature bag C

- Live-preview of the document
- Document outline preview
- Synchronized scrolling of the editor and preview
- Reorganizing sections in document using outline

The features are divided into three groups, based on their impact on the design and eventually implementation difficulties.

Group A are features causing the least of the problems. Though including non-trivial features to design or implement, they represent generic problems of source code editing, thus it is expected to be handled by an existing solution.

Feature bag B is more challenging for the implementation. The features likely require custom implementation tailored to the editor. While existing libraries can be used used in some cases (e.g. universal access for the Fullscreen API), more configuration and coding is required for all the features to affect the new editor component. Though there are not many features that are easy to implement, they

[44] Formally known as *markup editor*, which was a provisional title later scraped for being too generic. *Emily editor* name selected for (a) the acoustic resemblance to *"LML editor"*, (when spoken swiftly) and (b) it is a fancy name.

[45] This list is taken from [101] and reduced to exclude features which have been discarded for various reasons.

- have little effect on the other parts of the editor or future development and
- they still represent (more or less) existing problems, which have been solved in the past, though in different contexts.

The last bag is the most difficult. It contains the features that are very specific to the domain and related problems are likely original. As an example: the problem of the synchronized scrolling of the source code and its rendered counterpart is complicated and highly specific of the code nature. As was discovered in the research of the rival editors in [101], there are very few editors that offer this feature and if so, they are fixed and specialized on a single markup language. Emily editor tackles to provide this feature through general interface for the LMLs.

The solution to the features is discussed in the implementation chapter.

4.9.2 Emily's modules

The diagram 4.12 shows the main components and modules of the editor.

4.9.2.1 Components

The core component is the `Editor` itself, which includes the composite components `StatusBar` (the bar at the bottom), the `CommandPalette` and the `Outline` and utilizes supportive modules: `commands` for the `CommandPalette`, `autosave` and `lineNinja` definitions.

The `CommandPalette` has a supportive component for the drag&drop sorting.

4.9.2.2 Modes

The `modes` module includes the LML modes. There are two modes in the initial release. Since some functionality can be generated, all modes are bootstrapped before they are ready to be used in the editor.

4.10 Summary

The chapter provides design of the system in several perspectives including abstract concepts of core features, architecture schematics, module and component diagrams, API definitions and UI design. Though not covering all the components in the same level of detail, because of how vast the system is, the design solutions for the most crucial problems and core components are offered.

In course of the chapter, solutions for all the problems pointed out in the previous chapter *State-of-the-art* is provided:

- Uniform authorization is discussed in the *Repository providers*, where it is explained how to bind the application to the Gitolite through its CLI wrapper.
- Branching model support is discussed in the *RESTful API* section as well as in the *UI* section from the user perspective.
- Modular LML solution is proposed in *Emily editor*, where the core features are identified. This is further elaborated in the relevant implementation section.

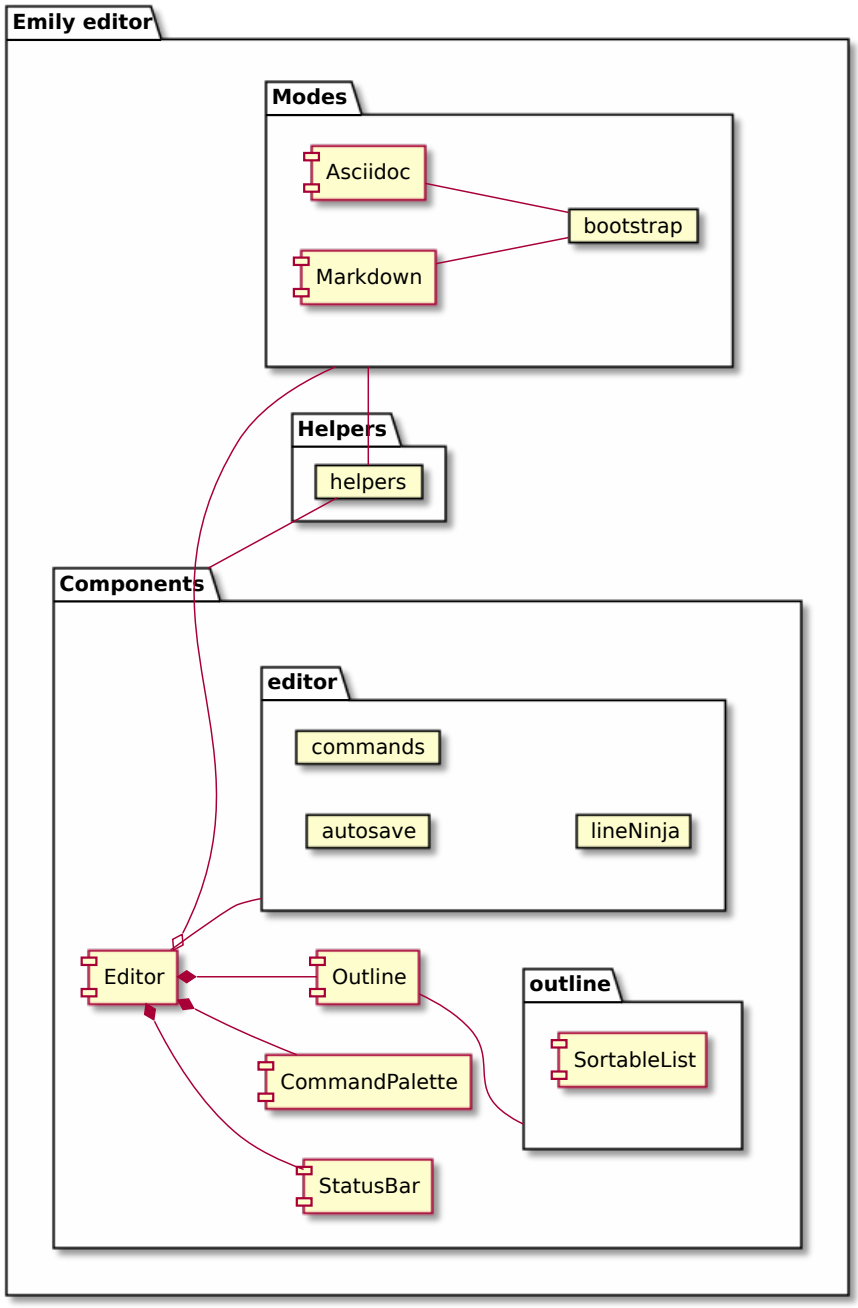

Figure 4.12: Design: Emily editor

Chapter 5

UI testing

In the chapter the wireframes designed in the previous chapter are subdued to a static form of usability testing – the heuristic analysis.

Jakob Nielsen's heuristics [68] is be used to analyze the existing wireframes. The 10 points of the heuristic are individually discussed on how the design holds up to them.

The names and brief descriptions are directly quoted from [68].

5.1 Analysis

1. **Visibility of system status** *"The system should always keep users informed about what is going on, through appropriate feedback within reasonable time."* [68]

The trying part of the system status are the pending changes. Though the system provides a summary through the commit modal, the status should be visible even without the extra interaction. At least in form of a binary indicator *clear – no changes* vs *modified – some pending changes*.

The repository index is missing the breadcrumbs menu. The breadcrumbs navigation should be available in the all major screens.

2. **Match between system and the real world** *"The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order."* [68]

The system uses terminology taken from Git documentation and the users are familiar with it through the Git CLI. The layout follows conventional guides, utilizing an application-wide navigation bar, context related optional side-bar menu and the main content area.

The usage of tabular menu for views of the file as well as for the actions upon it is confusing. The tabular menu should only hold the different read-only views of the file and have write-operations moved elsewhere.

3. **User control and freedom** *"Users often choose system functions by mistake and will need a clearly marked 'emergency exit' to leave the un-*

wanted state without having to go through an extended dialogue. Support undo and redo." [68]

This is achieved through the cumulative changes. The need of the user to commit is thus reduced to the bare minimum, when they can review changes before submitting.

4. **Consistency and standards** *"Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions."* [68]

Terminology is brief and, as stated, its terms reflect the Git lexemes.

5. **Error prevention** *"Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action."* [68]

There is an error potential in discarding the changes in the commit modal. Confirm prompt should be used.

6. **Recognition rather than recall** *"Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate."* [68]

The global UI is minimalistic from the perspective of the individual elements or menu items. Most of the navigation is formed by the content of the repository. The usage of UI elements is consistent and predictable.

The confusion arises from the repository index, which does not distinguish the repository's origins. As stated in design, the repositories can originate from the server or from a remote provider. This should be visible from the index.

7. **Flexibility and efficiency of use** *"Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions."* [68]

The system was designed with this feature at mind. It is mostly delivered through the direct SSH repository access and notable in the editor UI, utilizing the shortcuts and features seen in coding editors and IDEs.

8. **Aesthetic and minimalist design** *"Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility."* [68]

Only viable information is displayed. Some UI elements are compactly composed in order to diminish distraction, such as the use of the reference menu in breadcrumbs.

9. **Help users recognize, diagnose, and recover from errors** *"Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution."* [68]

No error messages are present in the current design.

10. **Help and documentation** *"Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large."* [68]

No form of user documentation is present and it should not be required since the systems UI reflects trends seen at popular services like GitHub.

5.2 Patching the wireframes

The list of the changes applied to the wireframes to mend the issues pointed out in the previous section follows. Each item includes a number of the figure with a corrected wireframe.

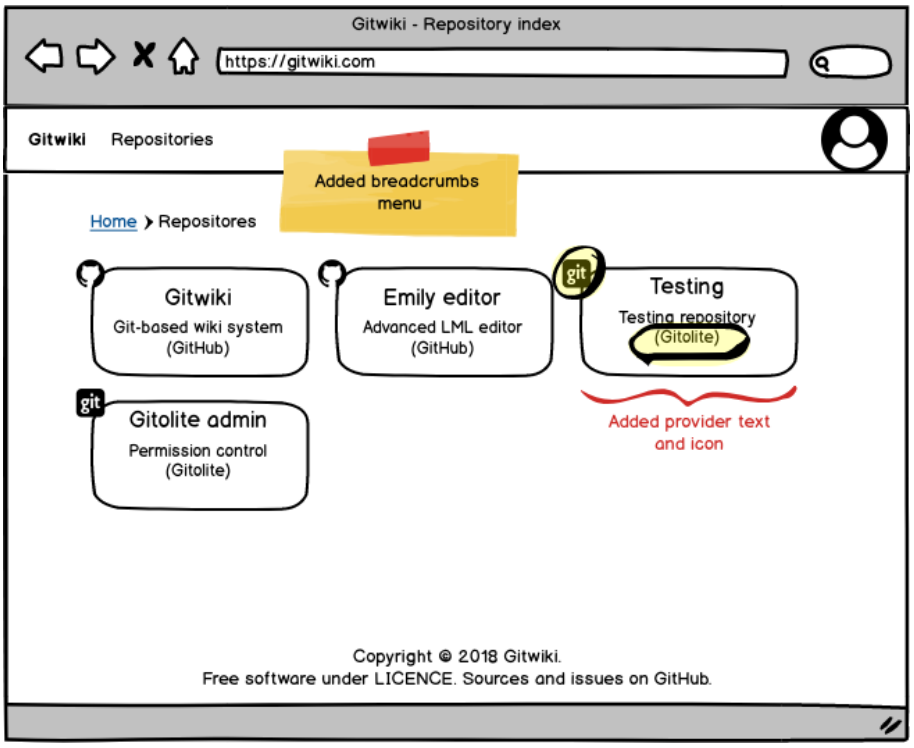

Figure 5.1: Wireframe: Repository index after heuristic analysis

- Added breadcrumbs menu to the repository index – image 5.1

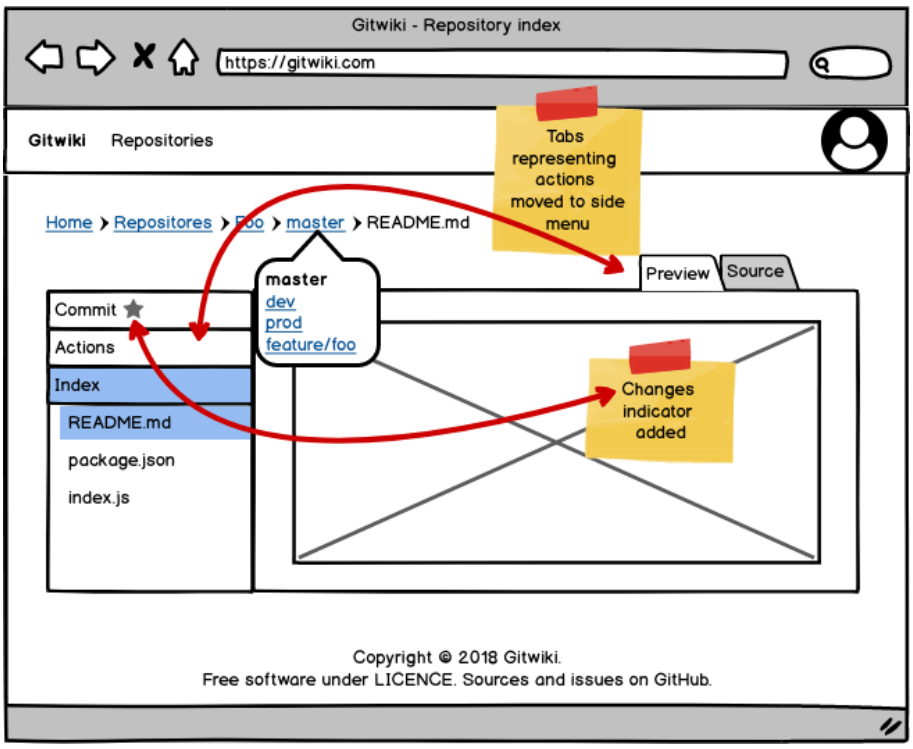

Figure 5.2: Wireframe: File preview after heuristic analysis

- Visually distinguishable (via icons) provider indication added – image 5.1
- Restructured the tabular menu and moved its items resembling actions into side menu – image 5.2
- Added a simple indicator for pending changes for the commit[46] – image 5.2
- Prompt confirm for discarding the changes in modal commit is required.

[46]Similar mechanism is used in Git CLI, when the CLI prompt is decorated to indicate there are pending changes in the working directory, index or in stash.

Chapter 6

Implementation

The implementation of the system is in the development for seven months, since October 2017. The source codes to this day sum up to 4 161 LOC and 770 commits. The detail stats are displayed in the table 6.1. The implementation is software, published from its early stages on GitHub.

The Emily editor source codes are separated from the main system for the potential of being used as an independent component, because of the rich and rather outstanding feature set mentioned in the design chapter. This lead to splitting the project into two distinct repositories:

- `grissius/gitwiki` [99] and
- `grissius/emily-editor` [97].

The repositories are published under the MIT [98] and the BSD 2-Clause [96] license respectively. Their user manuals and installation instructions are in appendices, along with their designed logotypes in the images F.1 and G.1.

The repositories are public – potentially anyone can contribute to the source code via PR. To the day of submission of the thesis however, the author is the sole contributor of the two projects.

Both projects have been developed with good manners at heart, thus both include:

- a readme file,
- a version file with current version of the project,
- and a changelog file following the *keepachangelog* [58] *standard*[47].

[47] Though being perhaps the only changelog guideline (thusly it is referred to it as standard) its format is more akin to best practices and reccomendation on how to write the changelog.

Table 6.1: Implementation repository statistics

Repository	lines of code	Commits	Releases
grissius/gitwiki	1 820	206	10
grissius/emily-editor	2 341	564	23

The projects also:

- adhere to *Semantic Versioning 2.0.0* [75] with published releases via Git tags on GitHub repositories,
- follow the Airbnb JS style guide [2] using ESLint and
- have been developed via a feature branch workflow.

The editor `grissius/emily-editor` is published on npm [100].

This introduction sums up the aggregate information about the implementation source codes. In the following section the notable dependencies used in the system are mentioned. In the rest of the chapter, instead of describing the development process as a whole, covers some of the major difficulties that appeared throughout the implementation process and describes the solutions provided for each one.

6.1 Used libraries

The major, design-defining technologies are already mentioned in the section *Technologies and tools* in the design chapter.

Apart from the libraries mentioned however, many other existing software is used for the implementation.

6.1.1 Source code editor

One of the most notable is Ace editor [3], which is utilized in the Emily editor as the underlaying, general purpose source code editor. It was used to replace the former CodeMirror [43] used in the prototype, because it was lacking a language mode for AsciiDoc.

Ace editor, apart from better performance, brings many advantages to the table, from which the most notable one is richer interface for the language modes[48], which allowed combining more than two modes at once[49] [50], clearer documentation and the mentioned native AsciiDoc support[51].

6.1.2 Libraries for development

From the development tools, ESLint is used for the coding style control, Babel [65] for JS compilation and Webpack [56] for the code bundling.

6.1.3 Other libraries

From the remaining notable software in the system's dependencies the following is used:

[48] Modules allowing to tag phrases in the content, typically for the purpose of syntax highlighting.

[49] CodeMirror allows to define only *mode* and secondary *backdrop mode*, which is used for e.g. spell-check.

[50] Ace provides context switching for modes, which works out of the box for selecting different highlighters for embedded code in Markdown). While this feature is also available in CodeMirror, the only example [42] demonstrates it on a XML file, using the `type` attribute to define context.

[51] This is missing in CodeMirror. Custom mode would need to be implemented.

- utility libraries Lodash [18] (in `emily-editor`) and functional JS library Ramda [55] (in `gitwiki`),
- Ant Design [6] React FE library and
- Pino [16] for logging.

6.2 UNIX permissions with Gitolite

As discussed in the analysis, Gitolite requires a single UNIX user to operate with. The remote users can use this Gitolite's user account to access the Git repositories using SSH. To prevent configuration errors and simplify the Gitolite setup process, it is encouraged to use a clean user account, with new UID for Gitolite.

Since application needs to access Gitolite CLI, installed in home directory of the user, and Gitolite recommends using a clean account, the following dilemma is confronted:

1. Either run the application under the Gitolite's UNIX user,
2. or run the application under any user, but solve the UNIX permissions.

The former represents the easier way – it is only required to install all application dependencies either globally or under this user. This brings two disadvantages:

- If the system already has another user set up to run similar applications, the administrator is required to possibly create redundancy.
- It pollutes the user with non-Gitolite data. This might not even unnecessarily complicate the Gitolite maintenance, but also, more importantly goes directly against the recommendations from Gitolite's manual. Disobeying the manual might discourage some users and also it would complicate the Gitolite installation, which itself is not trivial.

The latter is more complicated. When using another UNIX user, it requires an (executable) access to the Gitolite CLI program and also, all problems which arise from running the program in such manner solve, because it was not clearly designed with this scenario at mind.

The encountered issues are now discussed.

6.2.1 Run gitolite CLI under another user

With default installation, the CLI program, located at `/home/git/bin/gitolite`[52], is out of the box executable and lists valid help of the program.

However, when running the Gitolite with arguments that do something, e.g. `/home/git/bin/gitolite list-repos` an error occurs. The output of the operation under user *smolijar*[53] is in the listing 11.

[52]It is assumed Gitolite uses UNIX user *git*, as in installation manual, and its home directory is set according to the Gitolite installation manual as well, using the default location.

[53]This is a placeholder username used in the log files. In this section it refers to name of the account bound to Gitolite

```
1 FATAL: errors found but logfile could not be created
2 FATAL: /home/smolijar/.gitolite/logs/gitolite-2018-04.log: No such file
  ↪ or directory
3 FATAL: die  chdir /home/smolijar/.gitolite failed: No such file or
  ↪ directory<<newline>>
```

Listing 11: Implementation: Gitolite log error 1

The problem is seemingly banal. Gitolite plausibly utilizes the `$HOME` variable. It is creating logs in *smolijar's* home directory in a non-existent folder and tries to access the same folder. It should operate on the user it is configured with, in this case the *git* user. The output after setting the `$HOME` variable to `/home/git` is in the listing 12.

```
1 FATAL: errors found but logfile could not be created
2 FATAL: /home/git/.gitolite/logs/gitolite-2018-04.log: Permission denied
3 FATAL: cli  gitolite    list-repos
```

Listing 12: Implementation: Gitolite log error 2

With the `$HOME` variable updated, the Gitolite is successfully convinced to use the default directory. This however, brings another failure, which is an unknown error and insufficient permissions to log it. The unknown error might very possibly be caused by the same problem – the insufficient permissions to access Gitolite's files.

From the logs it is apparent and access to the *git's* home directory is required. At least with the write access for the `~/logs/` to successfully log the errors and the read access for the `/repositories`, which is most probably the cause of the unknown error.

The desired effect is to allow the access for another user, there are two options, setting permissions for user and for the group. Exposing any home directory *to all* users is an incredible security threat, even more so, considering the home directory holds the entrusted repositories. Thus setting permission for the group is the remaining solution.

6.2.1.1 Setup UNIX group

A UNIX group is created and setup in the following steps:

1. Create group *gitolite*: `sudo groupadd gitolite`
2. Add Gitolite and the current user to the group: `sudo usermod -a -G gitolite git && sudo usermod -a -G gitolite smolijar`
3. Change the *git* home repository's group ownership recursively: `sudo chgrp -R gitolite /home/git/`
4. Allow the user to write in selected folders: `sudo chmod -R 2775 /home/git/.gitolite`

6.2.1.2 SSH Secure mode

This is a side issue encountered when greedily setting the group's permission for the whole home directory `/home/git/` and not just the `.gitolite` sub-folder.

After setting up the permissions like so, Gitolite CLI seemingly works, while the Gitolite SSH interface stops working, rejecting all the connections with error regarding a missing repository.

"This is the default behavior for SSH. It protects user keys by enforcing `rwx------` *on* `$HOME/.ssh` *and ensuring only the owner has write permissions to* `$HOME`*. If a user other than the respective owner has write permission on the* `$HOME` *directory, they could maliciously modify the permissions on* `$HOME/.ssh`*, potentially hijacking the user keys,* `known_hosts`*, or something similar. In summary, the following permissions on* `$HOME` *will be sufficient for SSH to work."* [84]

SSH for security reasons kills any incoming connections to a users, whose home folder is by its standards insecure.

This can be bypassed by disabling the SSHD option *strict modes*[54]. This of course is dangerous and should not be performed on a machine, where the administrator does not have full control over the users, or cannot deny that a user with a configured remote access in the `authorized_keys` has a write access in their home directory. If Gitolite is set up properly and only provides authorized access via Gitolite CLI[55], it is be due its `command` option safe.

However, this is not required if setting the relaxed permissions only on the sub folder, as suggested in the previous section!

6.2.1.3 Owner of log files

Even when fixing the issue with SSH, later on, after both users have been using Gitolite for some time, yet another issue is nigh.

It is again a permission problem, and again with log files. The issue triggers the same error, as shown in 12, but this can happen for either of the users.

The problem is that *the other account*, in this case *smolijar*, creates a new log file, when the log files are swapped or new log is created. The log file is created with the correct GID (thanks to the `setguid` bit), by the user *smolijar*, but with the wrong permissions (no group access).

This is solved when the default set of permissions is set for the new files in the log folder. A similar problem is discussed in [78]. The desired effect can be achieved using the `setfacl` command:

`sudo setfacl -d -m g::rwx /home/git/.gitolite/logs/`[56]

```
1 # file: home/git2/.gitolite/logs/
2 # owner: git2
3 # group: gitolite
4 # flags: -s-
5 user::rwx
6 group::rwx
7 other::r-x
```

Listing 13: Implementation: Gitolite default ACL before

[54] Defining `StrictModes no` in `sshd_config`, usually located in `/etc/ssh/sshd_config`.

[55] As described in the analysis, Gitolite authorizes new users but instead of providing them with the full access, it only allows them to run the Gitolite program.

[56] The `-d` means *use default*, `-m` *modify* with argument.

The ACL settings can be displayed using the `getfacl <dir>` command. The results of before (listing 13) and after (listing 14) are present. Three new lines with default permissions are added. Now all the newly created log files by the user *smolijar* have the desired relaxed permissions for the group.

```
# file: home/git2/.gitolite/logs/
# owner: git2
# group: gitolite
# flags: -s-
user::rwx
group::rwx
other::r-x
default:user::rwx
default:group::rwx
default:other::r-x
```

Listing 14: Implementation: Gitolite default ACL after

6.3 Routes

This section tackles the problem of unified, scalable configuration of application's HTTP routing logic, and reusing the setup in BE and FE alike.

The routing in the application, needs to:

- bind routes on Express.js routers and
- provide navigation inside FE Next.js application.

It is a common practice to duplicate the string route definitions, which might be feasible for a small application, or one that does not utilize formatted URLs to this extent. Otherwise (in this case), the routing becomes unmaintainable as the application grows.

6.3.1 Independent routing logic

Having the BE and FE completely independent with each other is the easiest approach. The problem appeared when more than few routes that required formatting its arguments appeared. This required a refactor of the logic into an in-component helper functions as seen in the listing 15.

Since this is the first time the Next.js Link syntax is mentioned, it is briefly explained what the component does. Next.js provides implementation of the client-side navigation, when the application runs in the browser and takes care of the communication with the server. This is done not through a standard `<a>` anchor tag, but via a HOC `Link`.

The `Link` accepts (amongst others) the following React *props*[57]:

- **href**
  - This can be either a string, referring to the name of the page[58],

[57] React component's properties, are in the API documentation referred to as *props*.

[58] `repo/tree` loads the component in `pages/repo/tree.js`

```
import Link from 'next/link';

const link = pipe(
  concat('/repo/'),
  join('/'),
  filter(identity),
  props(['name', 'ref', 'path'])
);

// ...
const query = { name, ref, path };
const pathname = '/repo/tree'
const href = { pathname, query }
return(
  <Link
    href={href}
    as={link(query)}>
    <a>{name}</a>
  </Link>
);
```

Listing 15: Implementation: Generating routes via inline functions

  - or an object, as seen in the listing 15. The containing the page string under the key `pathname` and the query parameters in `query`.

- **as**
  - When using URL parameters, they are internally handled in the Next.js application through the query parameters. To use them in the URL, the definition of how the URL is going to look like in *as* property is required, in form of a string.
  - The *as* property only works in the client navigation. The Next.js application sets the document location to match the URL alias. However, this is just a visual facade for the client. All the communication with the server is handled via the former property, the `href`. The FE prompts the server for the e.g. `repo/tree?name=foo&ref=master&path=src`, no matter the alias.

This of course leads to a problem. If the user gets to the aliased URL not via the client navigation, but for instance by opening a shared link, the server responds with 404. The default Next.js handler, if alias URL is requested, e.g. `repo/tree/foo/master/src`, looks for page located in `pages/repo/tree/foo/master/src.js` by default logic and fails to find it, returning a *Not found* error.

This common issue is solved (as written the Next.js documentation) by creating the custom handlers, parsing the arguments from the URL and passing them to an appropriate Next.js render handler with the correct page parameter and query object.

This is already considered in the design, where FE router is included, which does exactly that.

6.3.2 Uniform route reference

Anyway, it is clear that the previous solution has some issues. Namely:

1. In-lining the `link` functions is not ideal for re-usability, since the same endpoint link is probably generated in several distinct components. It is be more appropriate to define the functions in separate module and import them at convenience into the components in FE.

2. As mentioned, Express.js route patterns need to be defined independently for custom BE handlers, delegating to Next.js handler. It is inconvenient to have Express.js and Next.js routing configuration separated, since the routes refer to the same thing.

For the stated matters the current solution is insufficient when operating with multiple routes, and code got more and more complicated.

Since there is no appropriate solution for the issue the following design solves the two issues.

```
1  const endpoints = {
2    TREE: 'TREE',
3    // ...
4  };
5
6  const routes = {
7    [endpoints.TREE]: {
8      generate: ({ name, ref, path }) => `/repo/tree/${[name, ref,
   ↪  path].filter(identity).join('/')}`,
9      express: '/repo/tree/:name/:ref/:path([\\S\\s]+)?',
10   },
11   // ...
12 };
13
14 exports.endpoints = endpoints;
15 exports.generate = endpoint => routes[endpoint].generate;
16 exports.expressPattern = endpoint => routes[endpoint].express;
```

Listing 16: Implementation: Routes module – definition

The route definition module example is in the listing 16. The user can access the endpoint constants and the `express` route definition and the `generate` function for the FE are side by side.

```
1  const { expressPattern, endpoints } = require('../../src/routes');
2
3  const router = express.Router();
4
5  router.get(expressPattern(endpoints.TREE), (req, res) => {
6    // ...
7  });
```

Listing 17: Implementation: Routes module – back-end

```
import Link from 'next/link';
import { endpoints, generate } from '../../src/routes';
// ...
const query = { name, ref, path };
const pathname = '/repo/tree'
const href = { pathname, query }
return(
  <Link
    href={href}
    as={generate(endpoints.TREE)(query)}>
    <a>{name}</a>
  </Link>
);
```

Listing 18: Implementation: Routes module – front-end

Using the module in BE is fairly easy and readable, as seen in 17.

How the shared route definition is used in the FE is shown in the listing 18.

6.3.3 Uniform route definitions

The previous solution using constants works well for creating an abstraction for the endpoints and places the definitions next to each other, making the code more organized.

There is still room for improvement, however.

The listing 16 features a redundancy, though not painfully obvious. The `express` pattern holds the very same information as the function `generate`, only in different notation. A uniform notation of singleton record can be used to represent the route.

The redundancy is more obvious when working with static routes, as showcased in the listing 19, where the two records are literally identical, apart from one being a function the other the literal value itself.

```
const routes = {
  [endpoints.INDEX]: {
    generate: () => '/repo',
    express: '/repo',
  },
  // ...
};
```

Listing 19: Implementation: Routes module – definition of a static route

After a research it is discovered what package is used in the Express.js routing[59]. The Express.js has a function to parse the pattern and extract the parameters. The custom generate function is just the direct inverse of the parse function, which is provided by the same library. The package `path-to-regexp` is not only used [21] by Express.js, but moreover it provides the desired function

[59]This is not default JS regular expressions syntax, though it resembles it. JS `RegExp` does not have a support for the named capture groups.

`compile`, an inverse to `parse`. All `generate` function are thus redundant, obsolete and can be generated with help of this library.

The difference is obvious from the definition in the listing 20, where the impact is the most drastic, removing the duplicate isomorphic definitions.

```
const endpoints = {
  front: {
    tree: '/repo/:provider/:name/tree/:ref/:path([\\S\\s]*)?',
    index: '/repo',
    // ...
  },
};
exports = endpoints;
```

Listing 20: Implementation: Routes uniform definition module – definition

The usage of the new route definition in the BE is almost identical, the wrapper function disappeared, returning the `express` pattern from the endpoint, as seen in the listing 21.

```
const { front } = require('../../common/endpoints');

const router = express.Router();

router.get(front.tree, (req, res) => {
  // ...
});
```

Listing 21: Implementation: Routes uniform definition module – back-end

On the FE, all the missing `generate` functions are substituted with a single `compile` function form the package `path-to-regexp` as seen in the listing 22.

```
import Link from 'next/link';
import { compile } from 'path-to-regexp';
import { front } from '../common/endpoints';
// ...
const query = { name, ref, path };
const pathname = '/repo/tree'
const href = { pathname, query }
return(
  <Link
    href={href}
    as={compile(front.tree)(query)}>
    <a>{name}</a>
  </Link>
);
```

Listing 22: Implementation: Routes uniform definition module – front-end

6.4 NodeGit

Whilst building the `git` module in Gitwiki BE application NodeGit is used.

In this section one part of the interaction with Git repository is discussed The interaction is retrieving a repository.

6.4.1 Get repository

In the `git` module, a dead simple API: *Get a repository* is desired. This of course needs some parameters that are provided by the repository provider:

- URL of the repository,
- FS destination path and
- authentication data.

While the former two can surely be strings, the last is more complicated. NodeGit has a class `Cred` [10] for representing the user identity.

6.4.2 Credentials

Generally `Cred` is used in all interactions inside a callback function which can react to the used username and the URL. Example usage of the credential callback, when setting options for cloning a repository, is seen in the listing 23 (the listing is taken from [11]). NodeGit thus provides an abstraction for the last item of complex type.

```
1 cloneOptions.fetchOpts = {
2   callbacks: {
3     credentials: function(url, userName) {
4       return NodeGit.Cred.sshKeyFromAgent(userName);
5     }
6   }
7 };
```

Listing 23: Implementation: NodeGit – Credentials callback

6.4.3 Function `getRepo`

The implementation of the function `getRepo` is discussed step by step and all the problems on the way are resolved.

The function is seen in the listing 24. It takes all the discussed parameters. The clone options object is created from the credential callback on the second line and setup (curried function) is created from it.

Then the cloning itself is performed, delegated to the NodeGit library, which returns a Promise with the repository or error.

If the cloning succeeds, the repository needs to be set up with the prepared function and `createLocalRefs` is called, which is discussed in a moment, and result is returned.

If the cloning fails, the encountered error is returned, unless it is the error code `EEXISTS`, which indicates that the repository could not have been cloned,

```
const getRepo = (uri, dest, getCred) => {
  const cloneOpts = getcloneOpts(getCred);
  const setup = setupRepo(cloneOpts);
  return NodeGit.Clone(uri, dest, cloneOpts)
    .then(setup)
    .then(createLocalRefs)
    .catch((e) => {
      if (e.errno === NodeGit.Error.CODE.EEXISTS) {
        return retrieveCachedRepo(dest, setup);
      }
      throw e;
    });
};
```

Listing 24: Implementation: NodeGit – Getting a repository

since the destination path points to a non-empty directory. This happens rather often, since the repository is often cloned for the first time only and then the cached local mirror is accessed on consecutive queries. On this error the repository is retrieved and updated it in the function `retrieveCachedRepo`.

6.4.4 Function `createLocalRefs`

The existence of the function requires a comment, even for the people using Git CLI on their daily bases. When cloning a remote, all remote branches are stored in the local references[60]. If the remote repository has more branches, all are correctly transfered and saved, but only the default branch (`master`) is created as a *local branch*[61]. To these branches user can checkout[62], but they cannot checkout in other references cloned from the origin, since they are not *branches* per se.

This is very much possible in Git CLI however. Though the *local branch*[63] does not exist, user can indeed `git checkout <branch>` to a branch that *only exists* in the remote references in Git CLI. This is just a syntax sugar for creating a head reference on the same OID as the remote reference; which Git CLI does for the user, on the first checkout into a branch that does not exist, but has a counterpart in the remote references of the same name. That is the reason why the line between remote references and head references is blurred for even advanced users of Git.

To finally get to the bottom of the function `createLocalRefs`, it exactly solves the discussed issue. Since there is no Git CLI behind NodeGit to create the head references, when they are needed, it is required to create them manually. The function is in the listing 25.

At first, all available references are retrieved from the repository, from which are filtered only the remote references. Then for each remote reference, the following actions must be performed:

[60] e.g. `.git/refs/remotes/origin/master`
[61] e.g. `.git/refs/heads/master`
[62] *Checking out* refers to setting the `HEAD` reference on a *branch* – not commit or tag; nor checking out files. Git terminology might be a little confusing at times overusing this word.
[63] Reference in `.git/refs/heads`

```
1  async function createLocalRefs(repo) {
2    const references = await
   ↪  repo.getReferences(NodeGit.Reference.TYPE.LISTALL);
3    const remoteRefs = references.filter(r => r.isRemote());
4    return Promise.all(remoteRefs.map((remoteRef) => {
5      const oid = remoteRef.target();
6      const upstreamName = getRefCompoundName(remoteRef.toString());
7      const { name } = parseRefName(remoteRef.toString());
8      return getOrCreateBranch(repo, name, oid)
9        .then(b => NodeGit.Branch.setUpstream(b, upstreamName));
10   }));
11 }
```

Listing 25: Implementation: NodeGit – Create local references

- Find the OID, so it is known onto which commit to *hook* the new branch[64]
- Get name of the remote reference (line 6) using a custom parsing function[65]
- Get the name of the branch[66] (line 7)
- Retrieve the branch (line 8)
  - Either get an existing branch (it might already exists in case of the second run or default branch),
  - or create it on the given OID
- Setup the remote reference as an upstream branch for the new local branch (line 9)

Setting up the remote is not necessary for using the branch locally, but for publishing it to the remote repository. The Git CLI user is familiar with the argument `--set-upstream` when pushing a branch to a remote for the first time. If the branch is created from the remote by Git CLI the upstream is automatically set[67].

6.4.5 Function `retrieveCachedRepo`

This function (its implementation is in the listing 26) is called with the destination, when the cloning fails due to an existing, non-empty destination folder. It needs to:

1. Create the repository abstraction using the NodeGit's `Repository.open`
2. Apply the provided setup method, created in and passed form the `getRepo` function
3. Update the head references with a set remote upstreams

[64] The OID is available through a synchronous method `target`, as seen on line 5 if listing 25.

[65] `remoteRef.toString()` returns the full path, e.g. `refs/remotes/origin/master`, while the NodeGit's API for creating a branch expects only the name of the remote, e.g. `origin/master`

[66] Ditto, prefix must be removed, converting `refs/heads/master` to `master`

[67] Tested on git version `2.7.4`

```
async function retrieveCachedRepo(dest, setup) {
  const repository = await NodeGit.Repository.open(dest);
  return compose(updateRemoteRefs, setup)(repository);
}
```

Listing 26: Implementation: NodeGit – Retrieve cached repository

The first two require no further comment, unlike the reference update. A repository that has been cloned some time before is being accessed. To get the repository that is up to date, without re-cloning it, each local branch is *pulled* from its configured upstream.[68] This action logic is in the function `updateRemoteRefs`.

6.4.6 Function `updateRemoteRefs`

This method *pulls* for each local branch with configured upstream. While `pull` is indeed command in Git CLI, it is not available in libgit2 and eventually neither in NodeGit. *Pull* is a user abstraction and shortcut for the two consecutive commands: `fetch` and `merge`.

`Fetch` for change is actually a command from Git core library and it updates the remote references to match the remote. After that (to complete the *pull*), it is required to update the local references to match the fresh remote references. This is achieved through hard resetting branches. After the branches are reset, the function is done and it returns the repository.

```
async function updateRemoteRefs(repo) {
  await repo.fetchAll(repo.fetchOpts);
  await createLocalRefs(repo);
  const ups = await branchesAndUpstreams(repo);
  await Promise.all(ups.map(([br, up]) => NodeGit.Commit.lookup(repo,
  ↪  up.target())
    .then(ci => NodeGit.Reset.reset(
      repo,
      ci,
      NodeGit.Reset.TYPE.HARD,
      new NodeGit.CheckoutOptions(),
      br.toString(),
    ))));
  return repo;
}
```

Listing 27: Implementation: NodeGit – Update branches with remote upstreams

As seen in the listing 27 the function proceeds as follows:

[68] While this method (*pull before you do anything*) is heedlessly practiced by the majority of the users, as satirically pointed out by [67], here the cause is justified. When applied by a *user*, it is usually to minimize the risk of an update conflict when pushing to a remote. Here on the other hand, it is to gain access to the current data, even when utilizing this form of caching.

1. Fetch all remote references using NodeGit's `Repository.fetchAll`
2. Create local references through a function that has already been discussed
3. Get pairs of local branches and their upstreams
4. For each pair:
    - Retrieve the commit of the upstream
    - Hard reset the branch to the commit

6.5 Emily

This section solves the issues from the design chapter with an additional issue of event recursive invocation in the synchronized scrolling.

6.5.1 Solving the feature bag C

6.5.1.1 Live-preview of the document

Live preview is not an issue from the implementation perspective, but it creates a clear restriction on the LMLs that are supported: The language needs to have an in-browser solution for rendering the source markup into HTML. Majority, if not all LMLs do satisfy this condition, because they usually originate from the web domain.

6.5.1.2 Document outline preview

Displaying the TOC, based on the document headlines is a simple matter, provided that a tool for generating HTML is available. All that is required is to parse the HTML, select the heading tags and form a hierarchical structure.

The first issue is excluding headlines from the outline. This is required since AsciiDoc has this feature[69]. This lays a second requirement on the LML module, a function to decide for an HTML headings, whether it is excluded from the outline. Since AsciiDoc utilizes CSS classes to propagate the generic attributes to HTML, the function for the AsciiDoc module merely checks the existence of a `discrete` CSS class in the HTML heading.

It is expected of the outline to serve as a navigation as well. Upon clicking the heading is looked up in the source code.

The solution used relies on concept of *line ninjas*[70], which is designed for the synchronized scrolling. As a side effect it labels the HTML output with elements bearing the corresponding line number in the source code. With it, the line number can be extracted from the HTML heading and the source code lookup is trivial.

6.5.1.3 Synchronized scrolling of the editor and preview

Line ninja[71]

The idea behind this is to smuggle the ninjas, into as many lines of the source code as possible. The ninjas must comply with the following rules:

[69] It is achieved by adding a `discrete` attribute to the heading.

[70] Line ninja is a name for hidden elements in code that are traceable by machine, but invisible to user.

[71] The name was created while developing a prototype to simplify terminology.

1. Ninja is a string
2. Ninja contains an encoded number, representing its line number in the source code
3. When the HTML is rendered from the LML source code containing ninjas, each ninja is left intact by the transformation and remains the identical string to the ninja before the transformation
4. Ninja does not alter the LML – if ninjas are removed from the HTML, the result is identical to HTML acquired from source without including ninjas

Number four is the most difficult to implement, and it is impossible to solve generally for all LMLs.

Thus a function `safelyInsert` is required by the LML module. It takes two arguments, a source code line of the LML and a string content. It returns a string representing the line but including the given string. This function assures that the markup is never altered by this change (if the content was removed after the HTML transformation). The function is very difficult to implement even for given LML, therefore the editor is fault tolerant towards it and works even if the function does not cover all the cases[72].

Eventually this function `safelyInsert` is used for smuggling ninjas into the LML source code. This way, they can be found in the resulting HTML and the editor can detect the breakpoints of source code lines, the only thing that remains is hiding the ninjas in the preview, which can be achieved via CSS.

This allows for the two way synchronization and also solves the issue with heading lookup as mentioned in issue with outline.

Since the ninjas are used in the outline, it is necessary for it to function properly that the `safelyInsert` performs the insert on every heading line. This is the only requirement for the function.

6.5.1.3.1 An example of line ninjas For illustration, an example of the usage of line ninjas is demonstrated.

Assume a Markdown source code in the listing 28. This is the plain source. As mentioned, before converting the document, the ninjas are inserted using the `safelyInsert` function. Its result is in the listing 29. Not all ninjas are perfectly smuggled into the code, as apparent. This depends on the implementation of the LML module.

All inserted ninjas satisfy the stated conditions: none of the destroy the markup, all bear the number of the source line and all of them are kept intact, when converted into HTML. This can be verified in the resulting HTML in the listing 30. All that remains to be done, is converting the ninjas into HTML markup, which does not shatter the resulting document. Using regular expressions is sufficient in this case. All the ninjas are converted into tags as seen in the listing 31.

Even using hidden spans leaves tracks in the rendered result and contrived CSS rulse must be used to clean them. Example of the CSS is in the listing 32.

[72] If the function fails to plant the content in the line, it return the line only.

```
# Header

A paragraph

Second paragraph with styles *italic*, **bold**, and `monospace`.
↪  Itemized list follows:

* ein
* zwo
* drei
```

Listing 28: Implementation: Line ninjas – Markdown

```
# Header @@@1@@@
@@@2@@@
A paragraph @@@3@@@

Second paragraph with styles *italic*, **bold**, and `monospace`.
↪  Itemized list follows: @@@5@@@
@@@6@@@
* ein @@@7@@@
* zwo @@@8@@@
* drei @@@9@@@
```

Listing 29: Implementation: Line ninjas – Markdown with ninjas

```
<div class="markdown-body">
  <h1 id="header-1-">Header @@@1@@@</h1>
  <p>@@@2@@@
     A paragraph @@@3@@@
  </p>
  <p>Second paragraph with styles <em>italic</em>, <strong>bold</strong>,
↪  and <code>monospace</code>. Itemized list follows: @@@5@@@
     @@@6@@@
  </p>
  <ul>
     <li>this one @@@7@@@</li>
     <li>that one @@@8@@@</li>
     <li>the other one @@@9@@@</li>
  </ul>
</div>
```

Listing 30: Implementation: Line ninjas – HTML with ninjas

6.5.1.4 Reorganizing sections in document using outline

With sufficient LML abstraction and line ninjas, line number of the selected heading can be detected. The same applies for the previous or the following heading. Using this technique, the sections can be moved around without having further requirements of the LML mode.

```
<div class="markdown-body">
  <h1 id="header-1-">Header <span class="ninja">1</span></h1>
  <p><span class="ninja">2</span>
     A paragraph <span class="ninja">3</span>
  </p>
  <p>Second paragraph with styles <em>italic</em>,
  ↪  <strong>bold</strong>, and <code>monospace</code>. Itemized list
  ↪  follows: <span class="ninja">5</span>
      <span class="ninja">6</span>
  </p>
  <ul>
     <li>this one <span class="ninja">7</span></li>
     <li>that one <span class="ninja">8</span></li>
     <li>the other one <span class="ninja">9</span></li>
  </ul>
</div>
```

Listing 31: Implementation: Line ninjas – HTML with ninjas in tags

```
.ninja {
  display: inline-flex;
  visibility: hidden;
  width: 0;
  height: 0;
}
```

Listing 32: Implementation: Line ninjas – CSS

6.5.2 Synchronized scrolling loop

Imagine the editor component containing the subcomponents for the preview and the source-code editor as in the listing 33. The listing shows a body of the render method of the component.

```
return (
  <Preview
    onScroll={this.handlePreviewScroll}
    ref={/*...*/}
    dangerouslySetInnerHTML={__html}
  />
  <SourceCodeEditor
    onScroll={this.handleEditorScroll}
    ref={/*...*/}
    onChange={this.handleChange}
    defaultValue={this.state.raw}
  />
);
```

Listing 33: Implementation: Emily – components

This is a minimalistic, yet logically complete schema of the components with regard to the discussed issue. There is the `SourceCodeEditor` with a default

value and an on-change handler; and the `Preview` that contains inner HTML, since it needs to be set from a string acquired by the converting tool.

Both components are referenced by the higher component to access their DOM elements when performing the scroll and both also have an on-scroll handler.

The desired behavior of the on-scroll handler for `SourceCodeEditor`, is to find the editor's current line and scroll the `Preview` to match it. Vice versa for the other handler.

Scrolling any element in the DOM is possible through the changing of its attribute `offsetTop`. Setting it to zero scrolls on the very top and any positive integer sets the scroll offset in pixels. Setting the `offsetTop` however, triggers a scroll event, the same was as if it has been scrolled by a user.

The interactions are displayed in the diagram 6.1. It is assumed that the user interacts with the editor but the communication is symmetric in the other case.

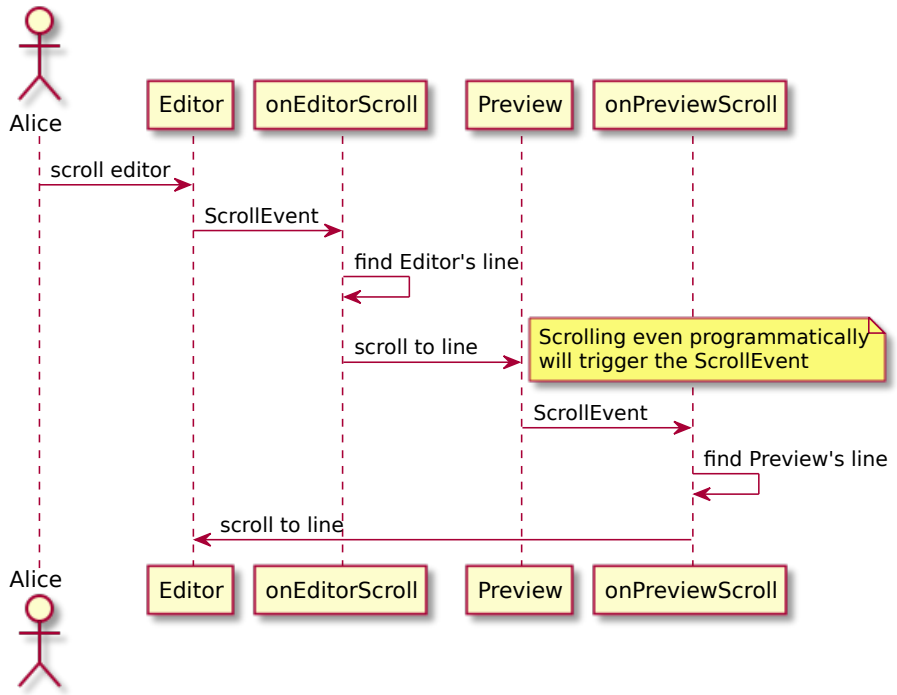

Figure 6.1: Implementation: Emily editor on-scroll listeners

1. The user scrolls the `Editor`, e.g. using mouse-wheel in the browser.
2. Scroll event on `Editor` is fired.
3. `onEditorScroll` is triggered.
4. `onEditorScroll` finds `Editor`'s line.
5. `onEditorScroll` sets `Preview`'s `offsetTop`.
6. Scroll event on `Preview` is fired.
7. `onPreviewScroll` is triggered.

8. `onPreviewScroll` finds `Preview`'s line.
9. `onPreviewScroll` sets `Editor`'s `offsetTop`.
10. *Repeat from point 3*

The loop theoretically runs forever. In practice it causes irritating scroll shivering momentum on the scrolled element.

The solution uses a ternary indicator with values:

- **editor**: *`Editor` scrolled last. It can scroll again but the `Preview` cannot.*
- **preview**: *`Preview` scrolled last. It can scroll again but the `Editor` cannot.*
- **clear**: *Anyone can scroll.*

The following rules apply to the listeners, regarding the indicator:

1. When the value allows you to scroll, execute and set to *your name*.
2. When the value forbids you to scroll, clear it and exit.
3. The default value of the indicator is *clear*.

An example interaction of how the circularity is broken is showed in the diagram 6.2, when the indicator's value is displayed in the notes.

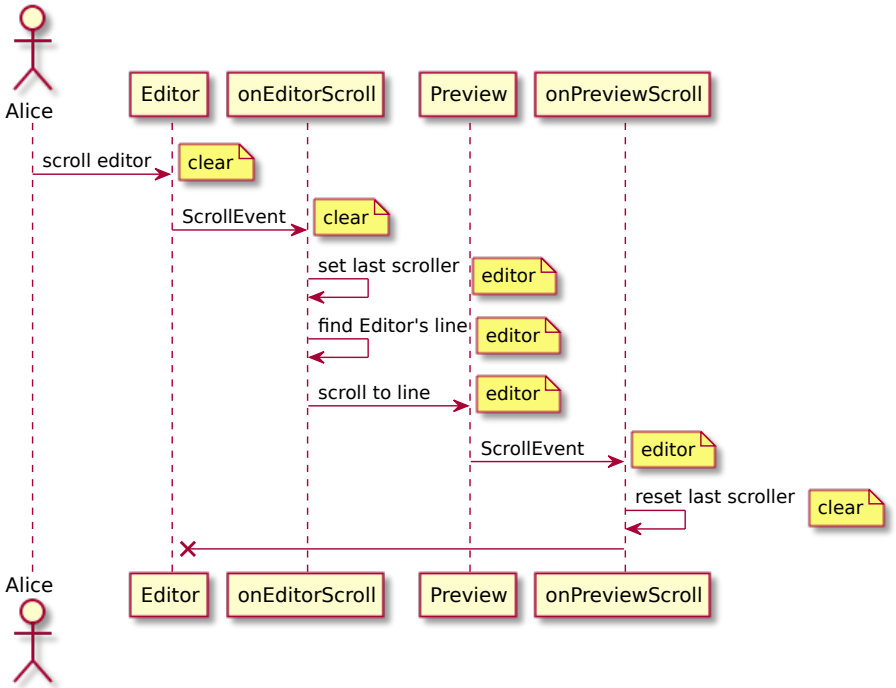

Figure 6.2: Implementation: Emily editor on-scroll listeners 2

The listing 34 shows the authentic implementation of the `handleEditorScroll` event listener. Lines 2 through 6 implement the indicator logic[73] After that a first visible line of the editor is computed the

[73] `null` stands for *clear* value

preview is scrolled using the method `scrollPreviewToLine`[74]. Before asking the editor for the line, it is prompted to reconfigure its renderer, which force updates the editor to react to the current scrolling event, allowing us to get an un-delayed line number.

```
1  handleEditorScroll = () => {
2    if (this.lastScrolled === 'preview') {
3      this.lastScrolled = null;
4      return;
5    }
6    this.lastScrolled = 'editor';
7    this.ace.renderer.$computeLayerConfig();
8    const firstVisibleLine = this.ace.renderer.getFirstVisibleRow() + 1;
9    this.scrollPreviewToLine(firstVisibleLine);
10 }
```

Listing 34: Implementation: Emily – editor scroll listener

Its counterpart implementation is in the listing 35.

It is very similar to the previous one, though bearing some differences. The `lastScrolled` indicator condition with the reset and the function's exit is present at the beginning of the function as in the previous case, but setting of the indicator is delayed. This is because at this point it is not sure that the scrolling is performed.

The editor is checked if it is scrollable in the direction. This solves the issue of scrolling out of bounds when preview is scrolled over the generated content beyond source, such as TOC, footnotes in appendix of the document etc. The source code editor provides an API to ask if it is scrollable by the given offset (line 10).

For this API it is required to know the direction of the scroll, which can be acquired by comparing it to the editor's current location[75].

If it is the case, the editor is scrolled and `lastScrolled` is properly set, otherwise the function ends.

[74] This function takes the line, computes the top offset in pixels and sets the appropriate attribute in the `Preview` element.

[75] At this point an idea to use the scroll event data to get the direction instead of comparing the lines might occur. Alas the event provides only the offset, not the delta, so the value would need to be subtracted one way or the other.

```
handlePreviewScroll = () => {
  if (this.lastScrolled === 'editor') {
    this.lastScrolled = null;
    return;
  }
  const firstVisibleLine = this.getPreviewFirstVisibleLine();
  const deltaPositive = firstVisibleLine >
↪   this.ace.renderer.getFirstVisibleRow() + 1;

  // dont scroll editor if preview scroll "out of source" (e.g.
↪   footnotes)
  if (this.ace.renderer.isScrollableBy(null, deltaPositive ? 1 : -1)) {
    this.lastScrolled = 'preview';
    this.scrollEditorToLine(firstVisibleLine);
  }
}
```

Listing 35: Implementation: Emily – preview scroll listener

Chapter 7

Testing

7.1 Automatic testing

For automatic testing of the application the Jest [23] testing framework with Enzyme [41], a testing utility library for React, and Chai [59], an assertion library for Node.js is used.

The `emily-editor` utilizes the test suits in its Travis CI pipeline before preforming a deploy of the demo application and publishing to npm.

7.2 Usability testing

Apart from heuristic analysis of the UI in the chapter *UI testing*, a live usability testing with working prototype and real users has been conducted.

In this chapter the testing scenarios are presented and then the testing itself and proposed solutions to UI issues are briefly summarized.

7.2.1 Testing scenarios

There are three short testing scenarios. As a whole the scenarios focus on the innovative aspects of the UI, that are unusual or unseen in similar projects.

The first scenario tests user's understanding of the parallel content browsing, forcing the user to change the revision of the repository and to read a file in the repository tree in a non-default branch.

The second scenario is designed to be more relaxing for the tester, inspecting the UI of basic navigation in the tree and the file detail.

The final scenario is the most challenging. Not only it is about the content editing but also it tests the concept of accumulating the pending changes in the application state, which is a feature not seen in any software mentioned in the chapter *State-of-the-art* nor in any other wiki software mentioned in the text.

7.2.1.1 Scenario A: Working with references

7.2.1.1.1 Introduction for the tester You are writing a user manual for a library, you and your team is developing. The manual is stored in the system in the repository `<repository>`. The project adheres to semantic versioning. Your

colleague has just fixed a bug in the installation section of the user manual that has caused many problems to the users and published it under the version `<version>`. Before deploying, check that the version of the project in the branch `<branch>` is greater than or equal to the `<version>`. The current version of the project is stored in the `./VERSION` file, apart from using Git tags.

7.2.1.1.2 Meta information

- **Expected time of completion:** 5 minutes
- **Goals:**
  - User can find the repository index.
  - User can find the desired repository in the index and open it.
  - User recognizes the reference widget in the breadcrumbs menu as means of navigation.
  - User understands that they can change the browsed reference using the widget and how it effects the view on the repository.
  - User can use the file index and select an item to bring up the file detail.
- **Initial state:** Homepage of the application with a logged in user
- **Terminal state:** Detail of the `./VERSION` file in branch `<branch>`

7.2.1.1.3 Steps

1. Navigate to the repository index.
2. Open the repository `<repository>`.
3. Find the file `./VERSION`.
4. What is the current branch?
5. Find the contents of the file `./VERSION` in branch `<branch>`.

7.2.1.2 Scenario B: Working with file detail

7.2.1.2.1 Introduction for the tester Your colleague forgot a "todo" note in a comment in one of the three document files in folder `<dir>`. The comment is on the first line of the file. Find the content of this note so you can create an issue in your tracker.

7.2.1.2.2 Meta information

- **Expected time of completion:** 5 minutes
- **Goals:**
  - User can navigate through the index menu to said folder.
  - User can understands that side-menu index is for switching between the files in the same folder.
  - User notices the tabular menu in the file detail, and can use it to select the view they desire.
- **Initial state:** Terminal state of the previous scenario or index page of the `<repository>` repository
- **Terminal state:** Detail of the source code of the commented file

7.2.1.2.3 Steps

1. Navigate to the `<dir>` in this repository
2. Browse the files in the folder to find the one with a "todo" comment note on the first line. Remember, that comments are not visible in the rendered document preview, but in the source code of the file.

7.2.1.3 Scenario C: Creating a revision

7.2.1.3.1 Introduction for the tester In the file you were just inspecting it is necessary to remove the comment and change the title of the document to `<newtitle>`. Apart from that, in the same revision, delete the remaining two files in the folder, they are no longer needed. Review your changes and create a commit with a message `<message>`.

7.2.1.3.2 Meta information

- **Expected time of completion:** 5 minutes
- **Goals:**
  - User understands the side-menu is context-relevant and contains actions related to the current screen.
  - User can navigate to page edit and add a change.
  - User realizes, that creating a change does not mean creating a revision.
  - User understands how the changes are accumulated in the application state.
- **Initial state:** Terminal state of the previous scenario
- **Terminal state:** Index of the repository `<repository>`

7.2.1.3.3 Steps

1. Remove the comment in the current file and change the title of the document to `<newtitle>`.
2. Delete the remaining files in the folder, except the one you have just edited.
3. Review all the pending changes.
4. Create a revision from the changes with comment `<message>`.

7.2.2 The course of the testing

Four UI testers participated in the testing in total.

No acceptance form inspecting the testers' background has been submitted. The testers were briefly introduces and thus it is known that they:

- are developers,
- know Git fairly well and use it regularly[76],
- know Markdown syntax, two users are also familiar with AsciiDoc, one of which prefers it to Markdown.

[76] except for one tester, who uses Subversion in their workspace, but uses Git on personal projects

All users qualified for the UI testing of the system. The UI of the system's prototype testing is conducted in combination of live testing and shared screen with voice chat. The testing provided only qualitative output in form a test log made during and after the testing.

7.2.3 Outcome

The following issues are discovered during the testing:

- The link in the repository index has incorrect cursor. *Solution: use pointer cursor to indicate the component is a link.*
- It is not apparent from the reference widget it is interactive. Its behavior surprises users. *Solution: add a caret icon to symbolize its function and use click event instead of hover to eliminate accidental interactions.*
- When browsing the repository tree and currently a folder is selected, duplicate file listing is visible in the main content as well as in the side-menu. *Solution: remove the index from the side-menu when on a folder.*
- In the side-menu, the listing of the current folder is titled *index*. Users stated that using label *files* is more self-explanatory. *Solution: change the label.*
- The users find it difficult to use the breadcrumbs menu for navigating to the root of the repository, since the menu has links for the Repository index, the current repository, the reference and path fragments. *Solution: visually divide the repository with reference from the path in breadcrumbs menu; remove repository index link.*
- SHA hashes seem to have different lengths using a proportional font. *Solution: use mono-space font for the SHA hashes.*

Conclusion

The goal of this thesis is to create a wiki system suitable for community software projects.

In the first chapter the goal is elaborated and core terms used in the text are explained. The users's needs are analyzed in the business process model and a viable solution for the problem of permission control with unified behavior across the UIs is proposed. With all the necessary data available, the system is defined through the requirements model. The functional requirements are further elaborated into the use-case model.

Acquiring the system definition in the analyses the existing wiki systems are reviewed. The systems are rated with regard to the raised criteria and their disadvantages are identified in the context of the intended use of the system.

The system is designed to either avoid these issues by the its nature or a solution is provided. The system design discusses its architecture, core components and UI.

The implementation chapter concludes the development process results and provides an in-depth view of selected problems faced during the implementation.

The tools used for automated testing are described in the testing chapter. The UI is tested for usability using Jakob Nielsen's heuristics [68] and in the final stage via conducted usability testing with users.

In the future, the system can be extended to provide pre-rendering of the repository's pages into HTML. However, this requires a thorough analyses and design of the solution to handle the current parallel development capabilities of the system; for instance caching only a single branch or a user or heuristic selected subset. Apart from this, the designed LML editor can be extended by other language modes, apart from the existing support for Markdown and AsciiDoc, or by richer user interactions inspired from IDE or coding editor development.

Bibliography

[1] Abramov, D.; et al. Redux [online]. April 2018, [Cited 2018-04-25]. Available from: https://redux.js.org/

[2] Airbnb, Inc. Airbnb: JavaScript Style Guide [online]. May 2018, [Cited 2018-05-01]. Available from: https://github.com/airbnb/javascript

[3] Ajax.org B.V. Ace – The High Performance Code Editor for the Web [online]. May 2018, [Cited 2018-05-01]. Available from: https://ace.c9.io/

[4] Allen, D. Textile Syntax Documentation and Sandbox [online]. April 2018, [Cited 2018-04-17]. Available from: https://txstyle.org/

[5] Allen, D.; White, S. Asciidoctor [online]. January 2018, [Cited 2018-01-31]. Available from: http://asciidoctor.org/

[6] Ant Financial. Ant Design: A UI Design Language [online]. May 2018, [Cited 2018-05-01]. Available from: https://ant.design/

[7] Atlassian. Bitbucket [online]. April 2018, [Cited 2018-04-05]. Available from: https://bitbucket.org

[8] Black Duck Software, Inc. Compare Repositories - Open Hub [online]. April 2018, [Cited 2018-04-04]. Available from: https://www.openhub.net/repositories/compare

[9] Branyen, T.; Haley, J.; et al. Install NodeGit [online]. April 2018, [Cited 2018-04-25]. Available from: http://www.nodegit.org/

[10] Branyen, T.; Haley, J.; et al. NodeGit: Cred [online]. May 2018, [Cited 2018-05-01]. Available from: http://www.nodegit.org/api/cred/

[11] Branyen, T.; Haley, J.; et al. NodeGit: SSH w/ Agent Guide [online]. May 2018, [Cited 2018-05-01]. Available from: http://www.nodegit.org/guides/cloning/ssh-with-agent/

[12] Chamarty, S.; et al. Gitolite – ad hoc user-created ("wild") repos [online]. April 2018, [Cited 2018-04-04]. Available from: http://gitolite.com/gitolite/wild/

[13] Chamarty, S.; et al. Gitolite – Performance [online]. April 2018, [Cited 2018-04-04]. Available from: http://gitolite.com/gitolite/perf/

[14] Chamarty, S.; et al. Gitolite – virtual refs (part 1) [online]. April 2018, [Cited 2018-04-04]. Available from: http://gitolite.com/gitolite/vref/

[15] Chamarty, S.; et al. Gitolite [online]. April 2018, [Cited 2018-04-04]. Available from: http://gitolite.com/gitolite/

[16] Collina, M.; Clements, D. M.; et al. Pino: Super fast, all natural JSON logger for Node.js [online]. May 2018, [Cited 2018-05-01]. Available from: https://getpino.io/

[17] Cudbard-Bell, A.; Vogt, C.; et al. Omnigollum: Omniauth authentication for gollum [online]. April 2018, [Cited 2018-04-17]. Available from: https://github.com/arr2036/omnigollum

[18] Dalton, J.-D.; et al. Lodash [online]. May 2018, [Cited 2018-05-01]. Available from: https://lodash.com/

[19] Dominik, C.; et al. Org Syntax (draft) [online]. April 2018, [Cited 2018-04-17]. Available from: http://orgmode.org/worg/dev/org-syntax.html

[20] Elouafi, Y.; Burzyński, M.; et al. Redux-Saga: An alternative side effect model for Redux apps [online]. April 2018, [Cited 2018-04-25]. Available from: https://github.com/redux-saga/redux-saga

[21] Embrey, B.; et al. Path-to-regexp [online]. April 2018, [Cited 2018-04-30]. Available from: https://github.com/pillarjs/path-to-regexp

[22] Facebook Inc. Flux: Application Architecture for Building User Interfaces [online]. April 2018, [Cited 2018-04-25]. Available from: https://facebook.github.io/flux/docs/in-depth-overview.html#content

[23] Facebook Inc. Jest: Delightful JavaScript Testing [online]. May 2018, [Cited 2018-05-01]. Available from: https://facebook.github.io/jest/

[24] Facebook Inc. Type Aliases | Flow [online]. April 2018, [Cited 2018-04-25]. Available from: https://flow.org/en/docs/types/aliases/

[25] Facebook Inc. Typechecking With PropTypes – React [online]. April 2018, [Cited 2018-04-27]. Available from: https://reactjs.org/docs/typechecking-with-proptypes.html

[26] Giard, N.; et al. Wiki.js | A modern open-source Wiki software [online]. April 2018, [Cited 2018-04-18]. Available from: https://wiki.js.org/

[27] Giard, N.; et al. Wiki.js: 2.0.0 - Dev [online]. April 2018, [Cited 2018-04-18]. Available from: https://github.com/Requarks/wiki#200---dev

[28] Giard, N.; et al. Wiki.js: git [online]. April 2018, [Cited 2018-04-18]. Available from: https://docs.requarks.io/wiki/install/git

[29] GitHub Inc. About GitHub Wikis, User Documentation [online]. April 2018, [Cited 2018-04-03]. Available from: https://help.github.com/articles/about-github-wikis/

[30] GitHub Inc. GitHub [online]. April 2018, [Cited 2018-04-05]. Available from: https://github.com/

[31] GitLab Inc. About Us | GitLab [online]. April 2018, [Cited 2018-04-04]. Available from: https://about.gitlab.com/about/

[32] GitLab Inc. File Locking | GitLab [online]. April 2018, [Cited 2018-04-04]. Available from: https://docs.gitlab.com/ee/user/project/file_lock.html

[33] GitLab Inc. GitLab CE [online]. April 2018, [Cited 2018-04-04]. Available from: https://github.com/gitlabhq/gitlabhq

[34] GitLab Inc. GitLab without gitolite | GitLab [online]. April 2018, [Cited 2018-04-04]. Available from: https://about.gitlab.com/2013/02/12/gitlab-without-gitolite/

[35] GitLab Inc. Protected Branches | GitLab [online]. April 2018, [Cited 2018-04-04]. Available from: https://docs.gitlab.com/ee/user/project/protected_branches.html

[36] Gogs. Gogs [online]. April 2018, [Cited 2018-04-04]. Available from: https://gogs.io/

[37] Gohr, A.; et al. Dokuwiki [online]. April 2018, [Cited 2018-04-17]. Available from: https://www.dokuwiki.org/dokuwiki

[38] Goodger, D. reStructuredText [online]. April 2018, [Cited 2018-04-17]. Available from: http://docutils.sourceforge.net/rst.html

[39] Google LLC. Google Docs [online]. April 2018, [Cited 2018-04-03]. Available from: https://docs.google.com/document/u/0/

[40] Gruber, J. Daring Fireball: Markdown [online]. January 2018, [Cited 2018-01-31]. Available from: https://daringfireball.net/projects/markdown/

[41] Harband, J.; Richardson, L.; et al. Enzyme: JavaScript Testing utilities for React [online]. May 2018, [Cited 2018-05-01]. Available from: https://github.com/airbnb/enzyme

[42] Haverbeke, M.; et al. CodeMirror: HTML mixed mode [online]. May 2018, [Cited 2018-05-01]. Available from: https://codemirror.net/3/mode/htmlmixed/index.html

[43] Haverbeke, M.; et al. CodeMirror [online]. May 2018, [Cited 2018-05-01]. Available from: https://codemirror.net/

[44] Hess, J.; et al. Ikiwiki: branches [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/branches/

[45] Hess, J.; et al. Ikiwiki: features [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/features/

[46] Hess, J.; et al. Ikiwiki: Free Software [online]. April 2018, [Cited 2018-04-17]. Available from: http://ikiwiki.info/freesoftware/

[47] Hess, J.; et al. Ikiwiki: git [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/git/

[48] Hess, J.; et al. Ikiwiki: gitbranch [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/templates/gitbranch/

[49] Hess, J.; et al. Ikiwiki: Hosting Ikiwiki with a master git repository on a remote machine [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/tips/Hosting_Ikiwiki_and_master_git_repository_on_different_machines/

[50] Hess, J.; et al. Ikiwiki: httpauth [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/plugins/httpauth/

[51] Hess, J.; et al. Ikiwiki [online]. April 2018, [Cited 2018-04-17]. Available from: http://ikiwiki.info/

[52] Hess, J.; et al. Ikiwiki: pagespec [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/ikiwiki/pagespec/

[53] Hess, J.; et al. Ikiwiki: separate authentication from authorization [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/todo/separate_authentication_from_authorization/

[54] Hess, J.; et al. Ikiwiki: wikiwyg [online]. April 2018, [Cited 2018-04-17]. Available from: https://ikiwiki.info/todo/wikiwyg/

[55] Hurley, M.; Chambers, D. Ramda Documentation [online]. May 2018, [Cited 2018-05-01]. Available from: http://ramdajs.com/

[56] Koppers, T.; Larkin, S.; Ewald, J.; et al. Webpack [online]. May 2018, [Cited 2018-05-01]. Available from: https://webpack.js.org/

[57] Kovitz, B. Creole: Cheat Sheet [online]. April 2018, [Cited 2018-04-17]. Available from: http://www.wikicreole.org/wiki/CheatSheet

[58] Lacan, O. Keep a Changelog [online]. April 2018, [Cited 2018-04-30]. Available from: https://keepachangelog.com/en/1.0.0/

[59] Luer, J.; et al. Chai [online]. May 2018, [Cited 2018-05-01]. Available from: http://www.chaijs.com/

[60] MacFarlane, J. Babelmark 2: Compare markdown implementations [online]. April 2018, [Cited 2018-04-18]. Available from: http://johnmacfarlane.net/babelmark2/

[61] MacFarlane, J. Pandoc – About pandoc [online]. April 2018, [Cited 2018-04-17]. Available from: https://pandoc.org/

[62] MacFarlane, J.; et al. Gitit: A wiki using HAppS, pandoc, and git [online]. April 2018, [Cited 2018-04-17]. Available from: https://github.com/jgm/gitit

[63] Mackall, M. Mercurial SCM [online]. April 2018, [Cited 2018-04-17]. Available from: https://www.mercurial-scm.org/

[64] McKay, J. Are there any statistics that show the popularity of Git versus SVN? – Software Engineering Stack Exchange [online]. April 2018, [Cited 2018-04-04]. Available from: https://softwareengineering.stackexchange.com/questions/136079/are-there-any-statistics-that-show-the-popularity-of-git-versus-svn/150791#150791

[65] McKenzie, S.; et al. Babel: The compiler for writing next generation JavaScript [online]. May 2018, [Cited 2018-05-01]. Available from: https://babeljs.io/

[66] Microsoft. Microsoft Word Online [online]. April 2018, [Cited 2018-04-03]. Available from: https://office.live.com/start/Word.aspx

[67] Munroe, R. xkcd: Git [online]. May 2018, [Cited 2018-05-01]. Available from: https://xkcd.com/1597/

[68] Nielsen, J. 10 Heuristics for User Interface Design: Article by Jakob Nielsen [online]. April 2018, [Cited 2018-04-27]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[69] Nieto, C. M.; Belfer, R.; Thomson, E.; et al. libgit2 [online]. April 2018, [Cited 2018-04-25]. Available from: https://libgit2.github.com/

[70] OpenBSD. sshd(8) – OpenBSD manual pages [online]. April 2018, [Cited 2018-04-04]. Available from: http://man.openbsd.org/sshd.8#command=%22command%22

[71] OpenBSD. sshd(8) – OpenBSD manual pages [online]. April 2018, [Cited 2018-04-04]. Available from: http://man.openbsd.org/sshd.8

[72] OpenID. OpenID Connect [online]. May 2018, [Cited 2018-05-06]. Available from: http://openid.net/connect/

[73] Oxford University Press. Word definition: wiki (noun), Oxford Advanced Learner's Dictionary [online]. April 2018, [Cited 2018-04-03]. Available from: https://www.oxfordlearnersdictionaries.com/definition/english/wiki?q=wiki

[74] Pornin, T. Is it possible to use a GPG or SSH key for web based authentication in a secure fashion? [online]. April 2018, [Cited 2018-04-05]. Available from: https://security.stackexchange.com/questions/44004/is-it-possible-to-use-a-gpg-or-ssh-key-for-web-based-authentication-in-a-secure/44013#44013

[75] Preston-Werner, T. Semantic Versioning 2.0.0 | Semantic Versioning [online]. April 2018, [Cited 2018-04-30]. Available from: https://semver.org/spec/v2.0.0.html

[76] Roundy, D.; et al. Darcs – Ideas/Branches [online]. April 2018, [Cited 2018-04-17]. Available from: http://darcs.net/Ideas/Branches

[77] Roundy, D.; et al. Darcs: FrontPage [online]. April 2018, [Cited 2018-04-17]. Available from: http://darcs.net

[78] Sherman, C. How to set default file permissions for all folders/files in a directory? - Unix & Linux Stack Exchange [online]. April 2018, [Cited 2018-04-29]. Available from: https://unix.stackexchange.com/questions/1314/how-to-set-default-file-permissions-for-all-folders-files-in-a-directory

[79] Sijbrandij, S. GitLab acquires Gitorious to bolster its on premises code collaboration platform | GitLab [online]. April 2018, [Cited 2018-04-04]. Available from: https://about.gitlab.com/2015/03/03/gitlab-acquires-gitorious/

[80] Software Freedom Conservancy. Git [online]. January 2018, [Cited 2018-01-31]. Available from: https://git-scm.com/

[81] Sørensen, J.; et al. Gitorious web interface built with Ruby on Rails [online]. April 2018, [Cited 2018-04-04]. Available from: https://github.com/gitorious/mainline

[82] Takezoe, N.; et al. Gitbucket [online]. April 2018, [Cited 2018-04-04]. Available from: https://github.com/gitbucket/gitbucket

[83] Thomas, D. RDoc – Document Generator for Ruby Source [online]. April 2018, [Cited 2018-04-17]. Available from: http://rdoc.sourceforge.net/

[84] Tillman, B. SSH and home directory permissions – Unix & Linux Stack Exchange [online]. April 2018, [Cited 2018-04-29]. Available from: https://unix.stackexchange.com/questions/37164/ssh-and-home-directory-permissions

[85] U.S. Department of Health & Human Services. Personas | Usability.gov [online]. April 2018, [Cited 2018-04-04]. Available from: https://www.usability.gov/how-to-and-tools/methods/personas.html

[86] Wall, L.; Burke, S. M. Perlpod – perldoc.perl.org [online]. April 2018, [Cited 2018-04-17]. Available from: http://perldoc.perl.org/perlpod.html

[87] Wikimedia Foundation. MediaWiki [online]. April 2018, [Cited 2018-04-17]. Available from: https://www.mediawiki.org/wiki/MediaWiki

[88] Wikimedia Foundation. Wikipedia [online]. April 2018, [Cited 2018-04-03]. Available from: https://www.wikipedia.org/

[89] Wikimedia Foundation. Wikitext – MediaWiki [online]. April 2018, [Cited 2018-04-17]. Available from: https://www.mediawiki.org/wiki/Wikitext

[90] Wikimedia Foundation. Wiktionary [online]. April 2018, [Cited 2018-04-03]. Available from: https://www.wiktionary.org/

[91] WikiWikiWeb. Gollum Wiki [online]. April 2018, [Cited 2018-04-17]. Available from: http://wiki.c2.com/?GollumWiki

[92] WikiWikiWeb. WikiWikiWeb: Front Page [online]. April 2018, [Cited 2018-04-03]. Available from: http://wiki.c2.com/?FrontPage

[93] WikiWikiWeb. WikiWikiWeb: Welcome Visitors [online]. April 2018, [Cited 2018-04-03]. Available from: http://wiki.c2.com/

[94] WikiWikiWeb. WikiWikiWeb: Wiki History [online]. April 2018, [Cited 2018-04-14]. Available from: http://wiki.c2.com/?WikiHistory

[95] ZEIT, Inc. Next.js: Framework for server-rendered or statically-exported React apps [online]. April 2018, [Cited 2018-04-25]. Available from: https://github.com/zeit/next.js/

[96] Šmolík, J. Emily editor: license [online]. May 2018, [Cited 2018-05-9]. Available from: https://github.com/grissius/emily-editor/blob/master/LICENSE

[97] Šmolík, J. Emily editor: React editor component for LMLs [online]. April 2018, [Cited 2018-04-30]. Available from: https://github.com/grissius/emily-editor

[98] Šmolík, J. Gitwiki license [online]. May 2018, [Cited 2018-05-9]. Available from: https://github.com/grissius/gitwiki/blob/master/LICENSE

[99] Šmolík, J. Gitwiki [online]. April 2018, [Cited 2018-04-30]. Available from: https://github.com/grissius/gitwiki

[100] Šmolík, J. npm: emily-editor [online]. April 2018, [Cited 2018-04-30]. Available from: https://www.npmjs.com/package/emily-editor

[101] Šmolík, J.; Uhnák, P.; Špak, M.; et al. Markup editor: Lo-fi prototype, Hi-fi prototype and Testing. 2018, semestral project MI-TUR. Available from: https://github.com/grissius/markup-editor-ui

Appendix A

Glossary

404 HTTP Not Found.

Angular 2 TypeScript-based open-source front-end web application platform.

Ember Open-source JavaScript web framework, based on the Model–view–viewmodel (MVVM) pattern.

ESLint Pluggable and configurable linter tool for identifying and reporting on patterns in JavaScript.

Express.js Fast, unopinionated, minimalist web framework for Node.js.

Flow Static Type Checker for JavaScript.

Flux Application architecture for building user interfaces.

Git Popular distributed VCS for non-linear workflow.

GitHub Web-based hosting service for version control using git.

local storage Window property which allows to save key/value pairs in a web browser.

Next.js Framework for server-rendered or statically-exported React applications.

NodeGit Asynchronous native Node bindings to libgit2.

NUR Návrh uživatelského rozhraní (Design of User Interface).

Pandoc Universal document converter tool.

Promise Object representing the eventual completion (or failure) of an asynchronous operation, and its resulting value.

PropTypes Type-checking definitions for React components.

React JavaScript library for building user interfaces.

Redux Open-source JavaScript library for managing application state.

textarea HTML element for multi-line input.

toolbar Row of symbols (icons) on a screen that show the different things that you can do with a particular program.

Vue Open-source JavaScript framework for building user interfaces.

wireframe Low-fidelity, simplified outline of the product's UI.

Appendix B

Acronyms

ACL Access control list.

API Application programming interface.

BE Back-end.

CGI Common Gateway Interface.

CI Continuous Integration.

CLI Command Line Interface.

CRUD Create Read Update Delete.

CSS Cascading Style Sheets.

DOM Document Object Model.

DTP Desktop publishing.

FE Front-end.

FS File system.

GID Group Identifier.

GNU GNU's Not Unix!.

GPL General Public License.

HOC Higher-Order Component.

HTML Hypertext Markup Language.

HTTP Hyper Text Transfer Protocol.

HTTPS Secure Hyper Text Transfer Protocol.

IDE Integrated Development Environment.

JS JavaScript.

JSON JavaScript Object Notation.

JSX JavaScript XML.

LML Lightweight markup language.

LOC Lines of code.

MVC Model-View-Controller.

OID Object Identifier.

OSS Open source software.

PR Pull Request.

REST Representational State Transfer.

RTE Rich text editor.

SCM Source Code Management.

SHA Secure Hash Algorithm.

SSH Secure shell.

SSHD SSH daemon.

SSR Server-Side rendering.

TOC Table of contents.

UC Use case.

UI User interface.

UID User Identifier.

UNIX Uniplexed Information Computing System.

URL Uniform Resource Locator.

UX User Experience.

VCS Version control system.

WUI Web user interface.

YFM Yaml front matter.

Appendix C

MI-NUR project highlights

In this chapter I shall showcase the lo-fi prototype design of the Emily editor from the MI-NUR project [101].

C.1 Acknowledgement

Contents of this section are separated from the main content, since *all figures* (diagram and wireframes) in this chapter are taken from the mentioned [101], which is a result of a teamwork.

The diagram and wireframes have been translated to English by myself for the purpose of the thesis.

I am the author of all texts in this chapter, which only briefly summarize or comment on the figures. If the reader is keen for more background information and development of the UI, they may read [101].

C.2 Task graph

The diagram C.1 displays transition of the editor states through relations between the UI screens.

C.3 Wireframes

C.3.1 The main view modes

The editor, as apparent from the diagram C.1, offers three display modes:

- Two column preview for common usage, wireframe C.2
- Source code for focusing on the content, wireframe C.3
- Preview for document revisions, wireframe C.4

C.3.2 Editor interactions

There are two wireframes showcasing the interactions with the editor. The first one, seen in figure C.5, demonstrates the main interface of the editor, the command palette, while the other shows all available navigational elements as seen in figure C.6

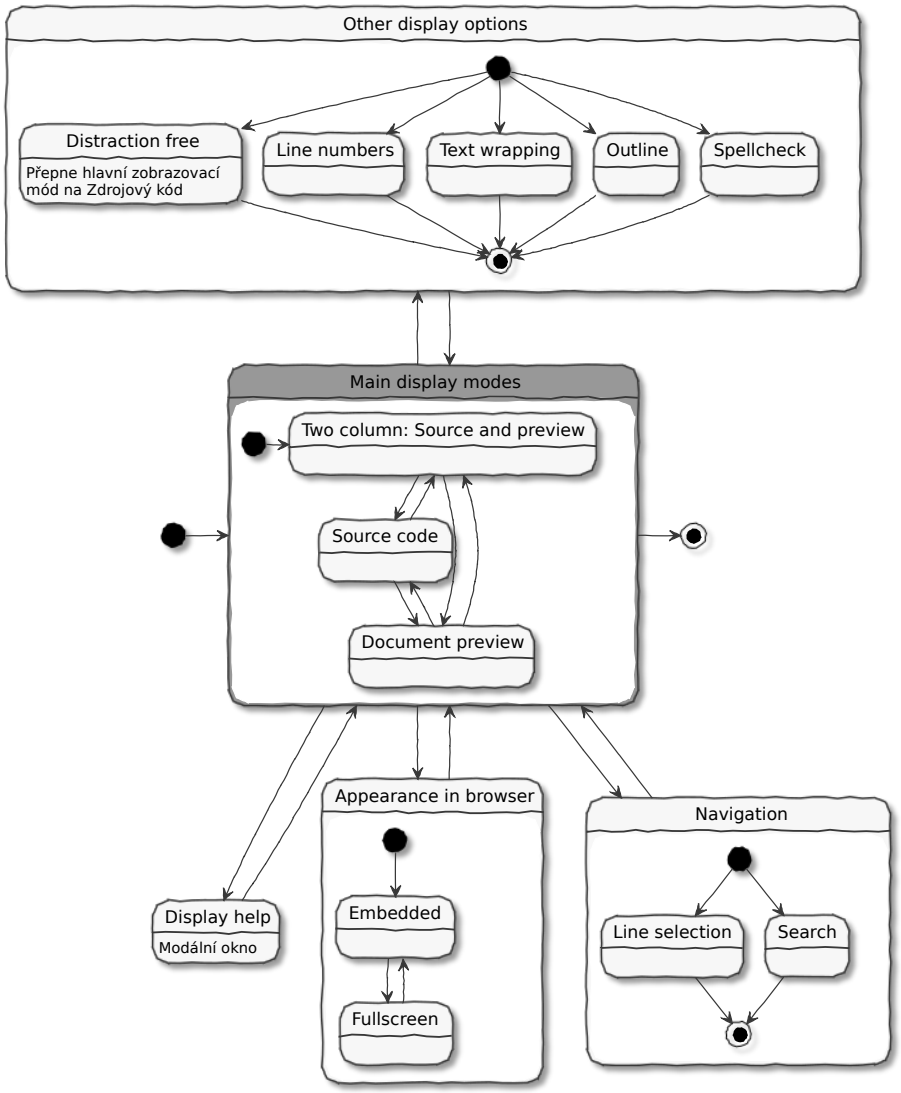

Figure C.1: Emily UI: Task graph

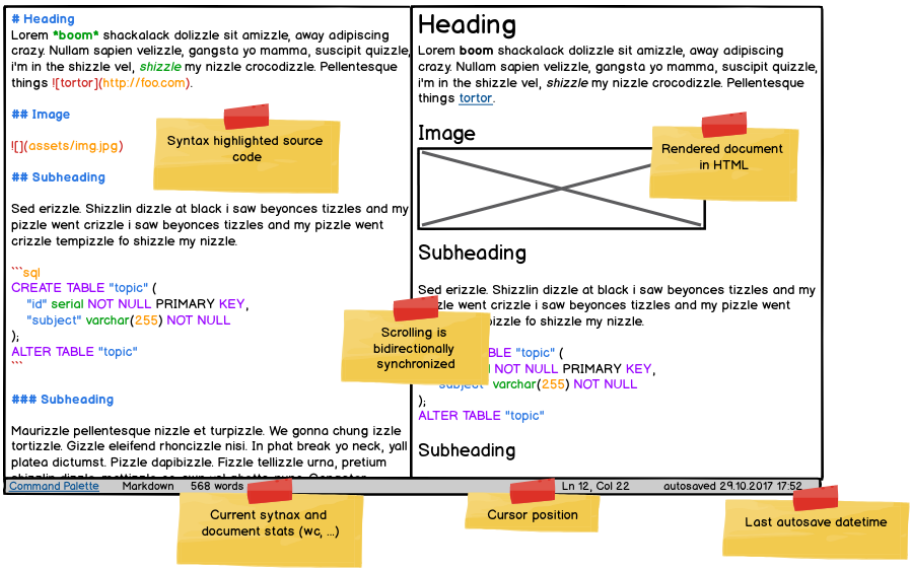

Figure C.2: Emily UI: Wireframe: Two column preview

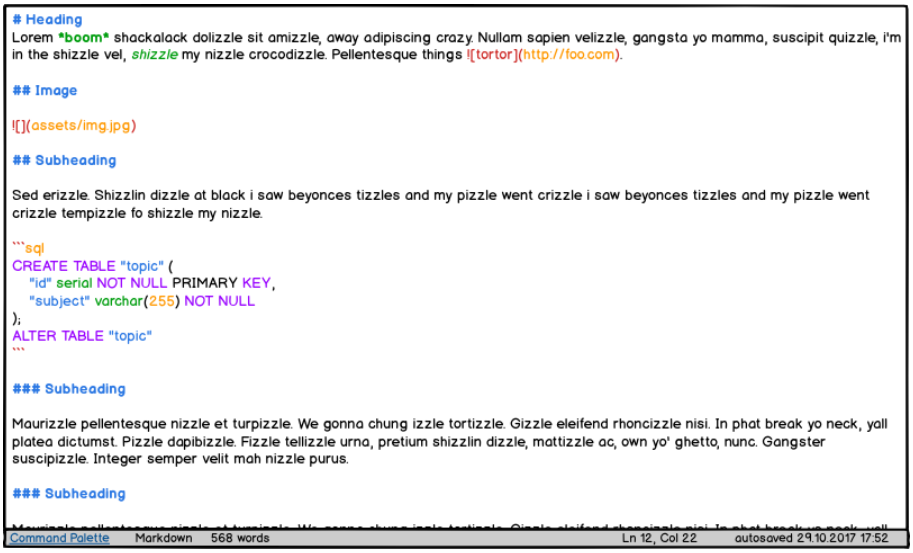

Figure C.3: Emily UI: Wireframe: Source code

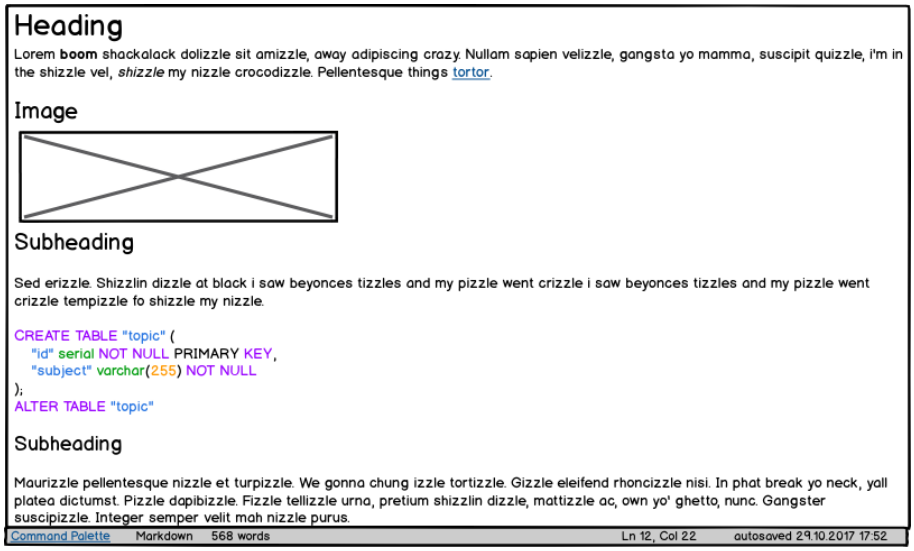

Figure C.4: Emily UI: Wireframe: Preview

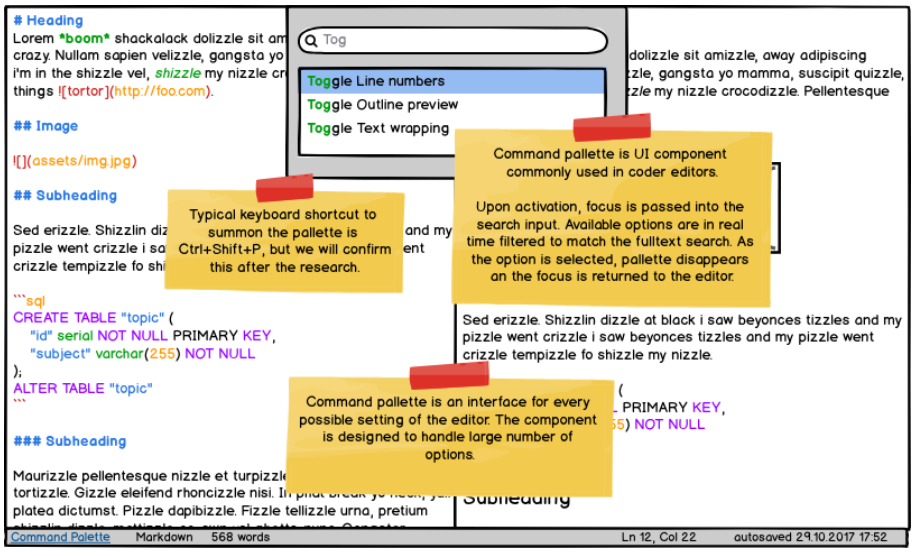

Figure C.5: Emily UI: Wireframe: Command palette

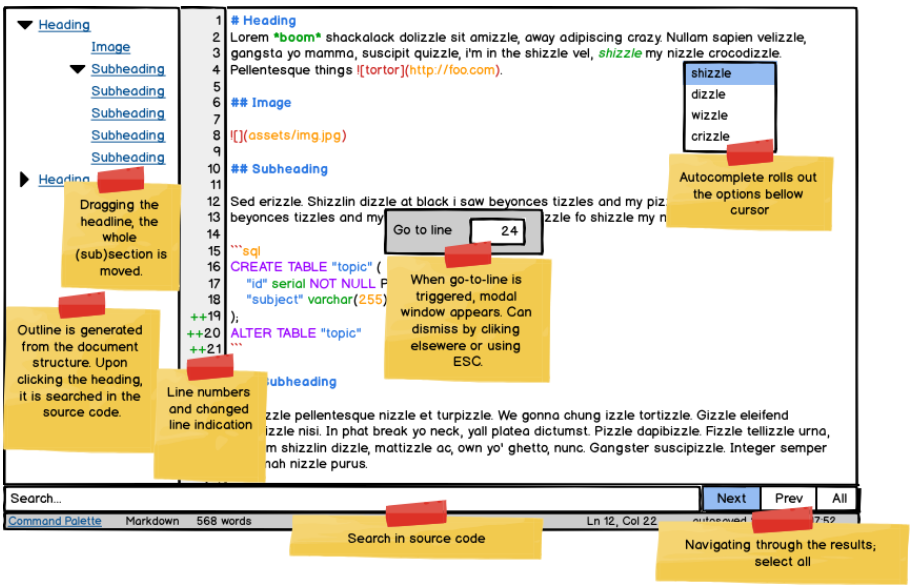

Figure C.6: Emily UI: Wireframe: Navigation

C.3.3 Display in browser

The editor is assumed to be by default an embedded editor component, seen on the figure C.7 and C.8.

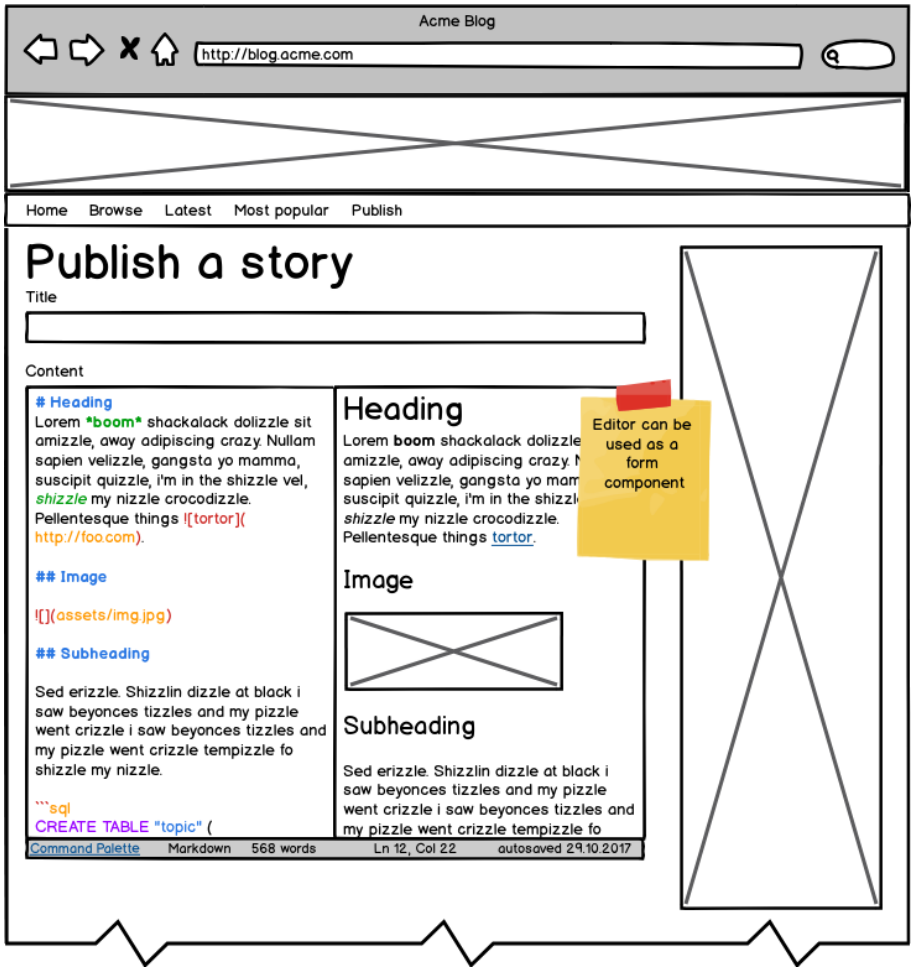

Figure C.7: Emily UI: Wireframe: Embedded

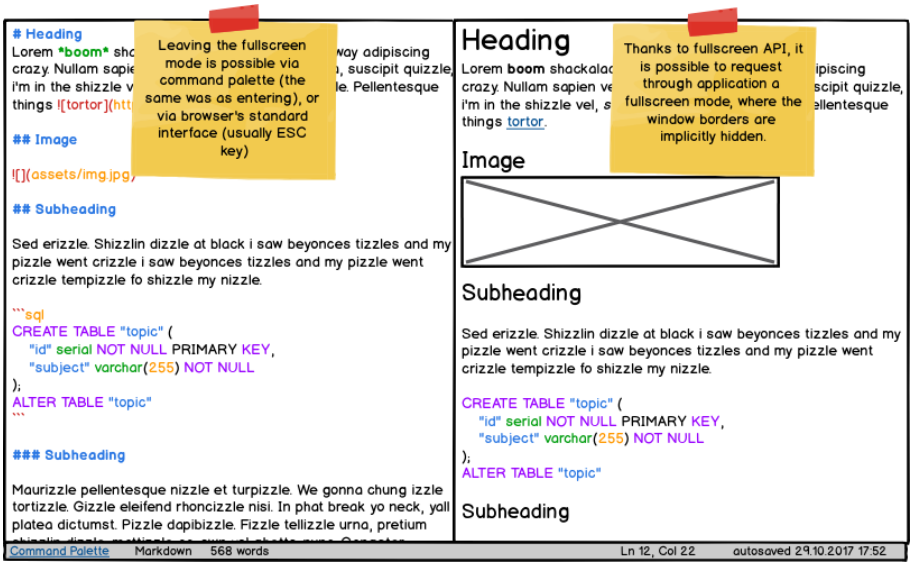

Figure C.8: Emily UI: Wireframe: Fullscreen

Appendix D

Gitwiki user manual

D.1 Gitwiki

Gitwiki is a git based wiki system with in-repository permission control, web user interface and Git CLI over SSH access.

D.2 About

It uses Gitolite authorization layer allowing complex, in-repository access control.

Gitwiki is part of an implementation for the Git-based Wiki System.

It uses Emily editor for document editing.

D.3 Install

The installation process is complicated, because repository hosting service over SSHd must be established.

D.3.1 Gitolite

D.3.1.1 Gitolite installation

This installation process is thoroughly explained here. Here is a step-by-step solution for Debian-based distributions.

Generate SSH keys:

```
# install git, sshd
sudo apt-get install openssh-server git
# generate a keypair for administration
ssh-keygen -t rsa -b 4096 -C "gitolite-admin" -f
 ↪  "$HOME/.ssh/gitolite-admin"
# copy the "~/.ssh/gitolite-admin.pub" for gitolite setup
cp ~/.ssh/gitolite-admin.pub /tmp
```

Install Gitolite:

```
# create new `git` user with home directory and set password
sudo useradd -m git
sudo passwd git

# switch to git user
su - git

# download and install gitolite
cd $HOME
git clone https://github.com/sitaramc/gitolite
mkdir -p bin
gitolite/install -to $HOME/bin # use abs path in argument

# setup gitolite with copied admin key from workstation
$HOME/bin/gitolite setup -pk /tmp/gitolite-admin.pub
```

D.3.1.2 Additional Gitolite setup

Wee need gitwiki to be able to access Gitolite. If you will be running gitwiki from a different user (assume username `jack`), you must perform additional setup.

Create group, add users *git*, *jack*, allow to write in `/home/git/.gitolite`

```
sudo groupadd gitolite
sudo usermod -a -G gitolite jack
sudo usermod -a -G gitolite git
sudo chgrp -R gitolite /home/git/
sudo chmod -R 2775 /home/git/.gitolite
```

Set `setgid` bit

```
chmod g+s /home/git/
```

Set default permissions for new log files

```
sudo setfacl -d -m g::rwx /home/git/.gitolite/logs/
```

D.3.2 Gitwiki

D.3.2.1 Install

D.3.2.2 Setup

D.3.2.2.1 Authentication

1. Register a new OAuth application
   - Set callback to `<host>/api/v1/auth/github/cb`
2. Remember `client_id` and `client_secret`

```
npm install
```

D.3.2.2.2 Configuration Create a `.gitwiki.config.js` and fill the data as in `.gitwiki.config.example.js`.

```
module.exports = {
  auth: {
    oauth2: {
      github: {
        // Information from the newly registered app
        client_id: '...',
        client_secret: '...',
      }
    }
  },
  gitolite: {
    // Path to gitolite bin
    bin: '/home/git/bin/gitolite',
    // Home directory of the gitolite's user
    home: '/home/git',
  },
  // Valid storage path for keyv(https://github.com/lukechilds/keyv)
  storage: 'sqlite:///home/git/database.sqlite',
};
```

D.4 Running

1. Add ssh key identity `ssh-add ~/.ssh/gitolite-admin` (path to private key you configured gitolite with)
2. `npm run start`

D.5 Usage

D.5.1 Repository providers

- Gitolite
- GitHub

To access GitHub repositories, you will be prompted to enter your *personal access token* in repository index. When provided, you can access your GitHub repositories apart from the default local (gitolite) repositories.

D.5.2 Adding SSH keys

The SSH keys are downloaded from GitHub on the first login and added to the Gitolite configuration. Apart from that, they can be added using Gitolite.

D.5.3 Permission control

This option is only available for Gitolite provider, for self-hosted repos. After a successful Gitolite setup, there is a repository `gitolite-admin`, where you can add users and chagne their permissions. If you are new to Gitolite, see Basic administration manual.

D.6 License

This project is licensed under the MIT license.

Appendix E

Emily editor user manual

Emily is a React editor component for LMLs, like Markdown or Asciidoc. The focus of the project is to provide fluent efficient interface for advanced users, who are familliar with using IDE or coding text editors.

E.1 About Emily

Emily is an editor for LML document formats, currently supporting few languages. Editor works with a document-format abstraction and new modules can be added to make use of existing features:

- Syntax highlight
  - Emily uses Ace editor under the hood, see supported languages
- Live document preview
  - Review the result as you type in split screen view or just browse the preview
- Outline preview
  - Section lookup in source code
  - Section reordering – drag & drop whole sections
- Command palette
  - Make use of a command palette you know from coding editors
- Autosave
  - Session is stored in localStorage, retrieved when lost.

Emily editor is part of an implementation for the Git-based Wiki System and its UI for the prototype has been developed in cource *UI Design* on the faculty.

E.2 Install

```
npm install --save emily-editor
```

E.3 Usage

1. Include `node_modules/emily-editor/dist/style.css`
2. Include `node_modules/emily-editor/dist/script.js`
3. Use component:

```
import Emily from 'emily-editor'
// ...

ReactDOM.render(
  <Emily />,
  document.getElementById('container')
);
```

For examples, see pages.

E.3.1 Props

E.3.1.1 `content`

Initial content of the editor

E.3.1.2 `language`

Language mode object. You can use `generateMode` to create a mode from existing modules.

```
import Emily, { generateMode } from 'emily-editor'
// ...

ReactDOM.render(
  <Emily language={generateMode(/*...*/)} />,
  document.getElementById('container')
);
```

E.3.1.3 `listFiles(pfx)`

List available relative files with path prefix `pfx`. Returns a `Array<String>` in a Promise.

This can be used for autosuggestions by a mode.

E.3.1.4 `width`

Lock editor's width and vorbid it to fill the container.

E.3.1.5 `height`

Lock editor's height and vorbid it to fill the container.

E.3.2 Methods

E.3.2.1 `getValue`

Return current value of the editor.

E.3.3 `generateMode(input)`

Input can be either:

- name of the mode, e.g. `asciidoc`
- any file path, e.g. `foo/bar/baz.adoc`

As a result a language mode is generated.

1. If the name or the extension matches an existing LML mode, a proper full-featured mode is generated.
2. If the name or the extension matches a mode supported by Ace editor, no special features for LML are provided, but editor features syntax highlight.
3. Otherwise a plaintext editor is delivered. No syntax highlight.

E.3.3.1 Examples

Here are some examples of using the editor with `generateMode` function.

```
// asciidoc mode
<Emily language={generateMode('x.adoc')}>
<Emily language={generateMode('asciidoc')}>
<Emily language={generateMode('/xxx/weee.adoc')}>

// markdown mode
<Emily language={generateMode('markdown')}>
<Emily language={generateMode('a/b/c/d/foo.md')}>

// (unsupported) js mode
// only syntax highlight, missing features
<Emily language={generateMode('javascript')}>
<Emily language={generateMode('test.js')}>

// unrecognized mode
// working in plaintext mode
<Emily language={generateMode('foo/bar/baz')}>
<Emily language={generateMode('thisisnotanameofanymode')}>
```

E.3.4 Language modes

Take a look at asciidoc mode example.

`name` (string) - name of the mode

`convert` (func) - converting function to html from the raw markup

`lineSafeInsert` (func) - insert content in the line of markup without distorting the markup - the more lines you can cover the better - it is necessary to cover heading lines

`postProcess` (func) - modify preview DOM before render

`renderJsxStyle` (func) - add styles for preview

`excludeOutlineItem` (func) - exclude DOM Element from the outline

`previewClassName` (string) - set the CSS classname for the prevew container

E.4 Online demo

https://emily-editor.herokuapp.com/

E.5 License

Emily editor is licenced under the BSD License.

Appendix F

Emily editor logo

Figure F.1: Emily editor logotype

Appendix G

Gitwiki logo

Figure G.1: Gitwiki logotype

Appendix H

Contents of enclosed CD

- `emily-editor` ..... the source codes of the LML editor
  - `README.md` ..... the readme file
  - `CHANGELOG.md` ..... the changelog file
  - `VERSION` ..... the version file
- `gitwiki` ..... the source codes of the gitwiki system
  - `README.md` ..... the readme file
  - `CHANGELOG.md` ..... the changelog file
  - `VERSION` ..... the version file
- `gitwiki-thesis` ..... the thesis text
  - `bin` ..... various scripts for thesis generation
  - `src` ..... the thesis Markdown source codes
    - `listing` ..... the listings used in the thesis
    - `assets` ..... static assets of the thesis
      - `diagram` ..... plantuml source codes of used diagrams
      - `images` ..... bitmap images
  - `Makefile` ..... make scripts for diagrams, pandoc, xelatex etc.
  - `bibliography.bib` ..... the bibliography file
  - `README.md` ..... the readme file
  - `CHANGELOG.md` ..... the changelog file
  - `VERSION` ..... the version file
  - `DP_Smolik_Jaroslav_2018.tex` ..... the thesis root LaTeX file
  - `DP_Smolik_Jaroslav_2018.pdf` ..... the thesis text in the PDF format