Third Year Project
Making Git Accessible For Children

Samuel Da Costa
Supervisor: Caroline Jay

April 10, 2019

Contents

Chapter 1

Introduction

In England and Wales, the percentage of students choosing to study a GCSE in computer science rose by 50% from 2015 to 2016 and continued to rise between 2016 and 2017 [14]. This is according to official statistics published by Ofqual.

Moreover, the UK government aims to introduce simple programming to children as young as five years old, suggesting that more and more young people are beginning to produce code.

This project does not aim to convince the reader that toddlers require a source control system, however for students at GCSE age, and for ALevel students, both of whom are completing projects involving non-trivial amounts of code, the need for comprehensive backups and version control become more apparent.

In the field of computer science, this is a solved problem. The *Git* source control solves all these problems. Git provides means of easily keeping backups, keeping track of versions and the ability to roll back to an old version easily.

Of course, using Git comes with its challenges. To use it effectively requires practice and an understanding of its low-level commands and can be daunting at first.

This project aims to strike a balance between the complexity of the system and the needs of younger users. To take advantage of such a rich version control system and to provide an interface to it which can be understood and used even by children.

The emphasis of this interface is on the concept of *version control* rather than Git specifically. Abstracting and automating the internals of Git to allow the user to concentrate on the higher level concepts of, "rolling back," and, "saving a new version".

1.1 Background on Git

This section briefly describes some of the main advantages of Git and introduce some relevant terminology.

As previously mentioned Git allows for straightforward backups. In practice, this means the existence of a remote server which also holds a copy of the user's code. When the user wishes to back up their code, they create what is known as a `commit`object. Such an object contains a subset of the files in the directory that the user has edited since the last commit, as well as metadata such as the date, author e.t.c. Importantly, the metadata contains a pointer to the commit that created before it. The resulting structure then forms a graph of project history.

At any point in the future, the user now can reset the project to the state stored in any `commit`. Given these objects, Git handles backups by providing commands that allow easiy storage of the objects on a remote server (referred to as *pushing* to the remote server).

Sharing of projects across devices now becomes easy. The user on a different device makes a copy of the project from the remote server (referred to as *cloning*). They are then free to create commits and to push them to the remote server, and to pull down commits created by others.

The reason Git is challenging to learn is a direct result of this structure. The following is merely the author's opinion and should not be interpreted as a statement of concrete fact.

This being said, it seems clear that Git's main complexity comes from the fact that any piece of code could be in one of many places. It could exist in a stash, in the working directory, in the staging area, in a commit on the remote e.t.c.

Users are required to know about all these different areas and how they interact. When they inevitably make a mistake, usually this results in an error which is very difficult to read and interpret.

With this in mind, the first goal of this project becomes clear. To abstract away as many of these different locations as possible. We will see that the project presents the user with only two areas: the remote version, and their local version.

1.2 Project Goals

On the outset of the project, there were two main objectives, from which all other aims extend. These are as follows.
Objective 1: To produce a user interface which should be useable by anybody over the age of thirteen years old. This interface should enable interaction with an underlying Git repository at a very high level of abstraction. *Note:* The GUI is not aiming to *teach* the user about Git, or even about version control. The idea, is that given a brief introduction to the concept of version control, that the user should be able to use the system.

What follows is the list of use cases the project initially aimed to fulfil. The use case highlighted in red was removed since it was judged to add needless complexity.

- Ability to login on any machine with the software installed and see a list of projects.
- Ability to open any of these projects locally just with a click.
- New commits should appear on other machines logged into the same user account without the need for the user to perform manual steps.
- Ability to jump back and forwards through the project history.

- Ability to choose which files to add to a commit

Objective 2 The second objective is less tangible and is best described as follows. To produce a system that is scalable, well tested and robust enough for use by inexperienced users. In practice, this last part implies that the system should be able to automatically detect and correct issues with the underlying Git repository which would usually need to be manually corrected.

Scalability is an essential part of this project and warrants further discussion. Consider the case that the project is wildly successful and has many users beginning to use it. The result would be many repositories needing storage on remote servers. One of the key aims of the project is to build a system that allows adding more remote machines to the pool of available resources and that the system automatically begins using these multiple servers.

b

Chapter 2

Technical Design

This chapter aims to describe and explain the system's overall architecture. To explore potential performance bottlenecks and their solutions, as well as discussing the roles of the individual components that make up the final solution. This chapter does not discuss the implementation details, but rather give a high-level view of the *why* rather than the *how.* It is relevant to mention that the system as a whole has been informally dubbed 'littlegit', a name used throughout this report.

2.1 Other work in this area

So-called *git guis* have existed almost as long as git itself has existed. Notable, git itself comes with two visual tools, **gitk** and **git-gui**. The former is used mainly for viewing the commit history, and the latter's main function is to create commits [10].

However, these tools provide very little functionality beyond a graphical way of invoking git commands. There is very little automation, and a reasonable knowledge of Git is still very much required. Furthermore, they provide no help with generation of SSH keys, which are used by Git for authentication.

At the time of writing the most comparable system to that which this project aims to create belongs to Atlassian. The user can manage remote repositories, clone them to their local machine, view history and interact with the repository in man complex ways through the desktop application *Source Tree.* The most noticeable feature here is the management of remote repositories from within the same application.

However, here too is where the automation stops. There is little to no help with the resolving of conflicts, SSH key setup is still a manual process, and most importantly, errors from git are often directly passed onto the user with no attempt to resolve the issue for them. Figure **??** is an example of this.

The project, therefore, aims to build upon the positive features of Source Tree and the Atlassian ecosystem, adding more abstraction and automation in the hopes of achieving a much simpler user experience.

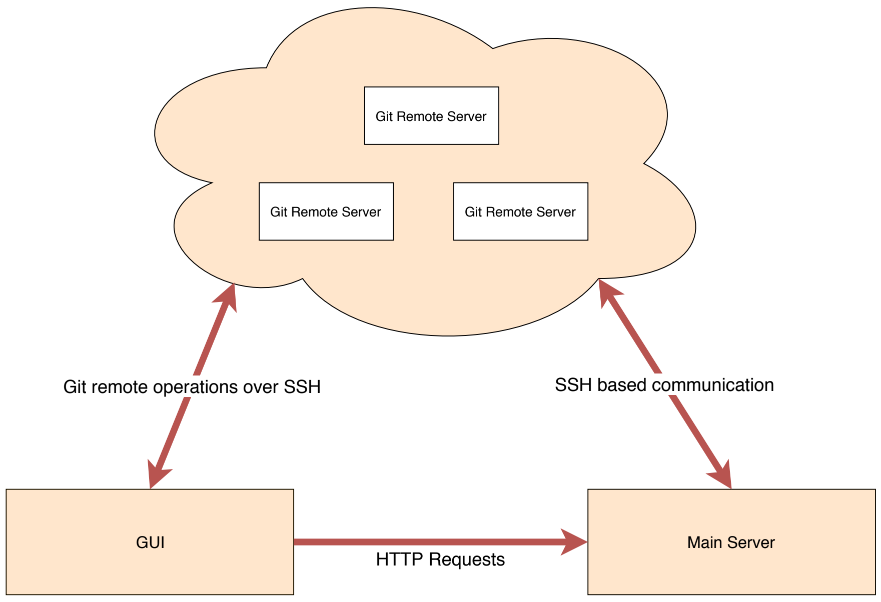


Figure 2.1: A high level representation of the architecture of the system.

2.2 Technical Overview

First, we present a very high-level description of the system as a whole, before exploring its components and their roles.

Figure 2.1 shows an overview of the system. The three main components shown run on distinct machines communicating over the internet. The GUI is the graphical user interface that runs on the user's machine, and what they use to manipulate their repositories.

The reader may recall that to take advantage of Git fully as a source control system the user must define a *remote repository*. A repository hosted on a remote server which remains in sync with the local copy on the user's machine. The Git servers in Figure 2.1 are the machines that host these remote repositories.

The final component is the main server. It does *not* host remote repositories but manages the system as a whole. The main server keeps track of users and their repositories, as well as ensuring that the correct users have access to the correct repositories.

2.2.1 Git Servers and the Main server

The reader may ask whether this infrastructure is needed. There already exist many hosting services for Git repositories. GitHub, Gitlab and Bitbucket to name a few. However, the issue with using these is our goal of automating as much as possible. To be able to perform tasks such as setting up SSH keys for the user we must have complete control over the backend infrastructure.

Furthermore, this control allows for easy integration of new features in future. For example, we could introduce a new type of user such as a teacher

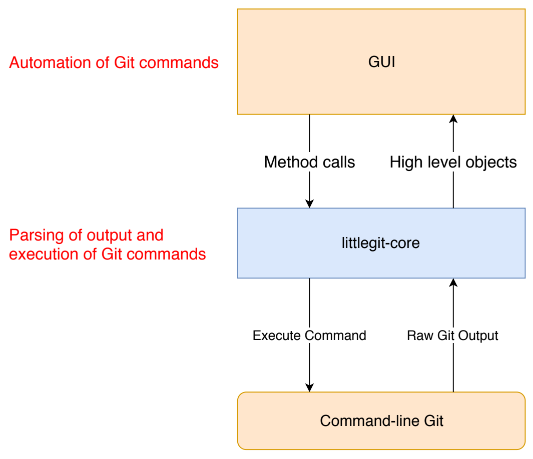


Figure 2.2: A representation of the interaction between the GUI and the littlegit-core

who could have access to multiple repositories. To introduce such a role while relying on the stability of a third party service would be much more difficult.

Figure 2.1 shows multiple Git servers; this demands some explanation. A reasonable question to ask is why is one not sufficient? The second of the two objectives for this project is to produce a scalable system. We must ask ourselves what happens if we start to get more users, creating more repositories. One server soon runs out of space. Moreover, our system is likely to slow when one Git server alone has multiple other machines communicating with it simultaneously.

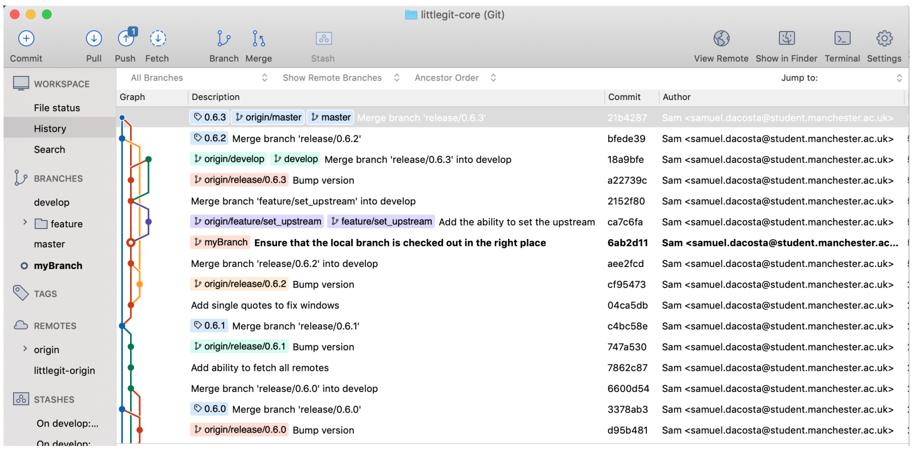
For this reason, we spread the load of Git repositories over multiple machines. A future goal of this project is to have machines hosted in different regions around the world, and assign their repositories based on the location of users to make the system as efficient as possible.

Of course, having multiple servers hosting Git repositories presents its own set of challenges. For a user's machine to be able to communicate with a Git server, we must register its public SSH key on all the Git servers that host the user's repositories. One of the responsibilities of the main server is precisely this. To manage SSH keys, ensuring the right ones exist on the correct machines, which becomes more complicated when we realise that an SSH key is associated to a machine, not a user and that a user may work on multiple machines necessitating multiple keys.

The management of SSH keys is only one role assigned to the main server; it also creates Git repositories on the Git servers and manages users, keeps track of their repositories and handles all user authentication.

2.2.2 GUI

The GUI is the desktop application the end user interacts with and uses. Its primary responsibilities are to allow the user to manipulate the Git repository and visually display its current state.



Description	Commit	Author
0.6.3 origin/master master Merge branch 'release/0.6.3'	21b4287	Sam <samuel.dacosta@student.manchester.ac.uk>
0.6.2 Merge branch 'release/0.6.2'	bfede39	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/develop develop Merge branch 'release/0.6.3' into develop	18a9bfe	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/release/0.6.3 Bump version	a22739c	Sam <samuel.dacosta@student.manchester.ac.uk>
Merge branch 'feature/set_upstream' into develop	2152f80	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/feature/set_upstream feature/set_upstream Add the ability to set the upstream	ca7c6fa	Sam <samuel.dacosta@student.manchester.ac.uk>
myBranch **Ensure that the local branch is checked out in the right place**	**6ab2d11**	**Sam <samuel.dacosta@student.manchester.ac...**
Merge branch 'release/0.6.2' into develop	aee2fcd	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/release/0.6.2 Bump version	cf95473	Sam <samuel.dacosta@student.manchester.ac.uk>
Add single quotes to fix windows	04ca5db	Sam <samuel.dacosta@student.manchester.ac.uk>
0.6.1 Merge branch 'release/0.6.1'	c4bc58e	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/release/0.6.1 Bump version	747a530	Sam <samuel.dacosta@student.manchester.ac.uk>
Add ability to fetch all remotes	7862c87	Sam <samuel.dacosta@student.manchester.ac.uk>
Merge branch 'release/0.6.0' into develop	6600d54	Sam <samuel.dacosta@student.manchester.ac.uk>
0.6.0 Merge branch 'release/0.6.0'	3378ab3	Sam <samuel.dacosta@student.manchester.ac.uk>
origin/release/0.6.0 Bump version	d95b481	Sam <samuel.dacosta@student.manchester.ac.uk>

Figure 2.3: A screenshot of the main interface of the application SourceTree.

Low-level git library

At this stage, it's prudent to mention the way both the GUI the main server interact with Git itself. This interaction happens through the medium of a library called the **littlegit-core** written for this system. It serves as a wrapper around the Git command line allowing interaction from a high-level language (discussed further in chapter 3) with the Git binary. The littlegit-core is responsible for executing git commands and parsing the output into useful high-level objects.

However, the reader should note that this layer performs none of the automation of Git which the system aims to achieve. This layer executes git commands then parses and returns the resulting output as Figure 2.2.

Both the main server and GUI include the littlegit-core library as a dependency, allowing them to interact with Git repositories without interacting directly with the Git binary.

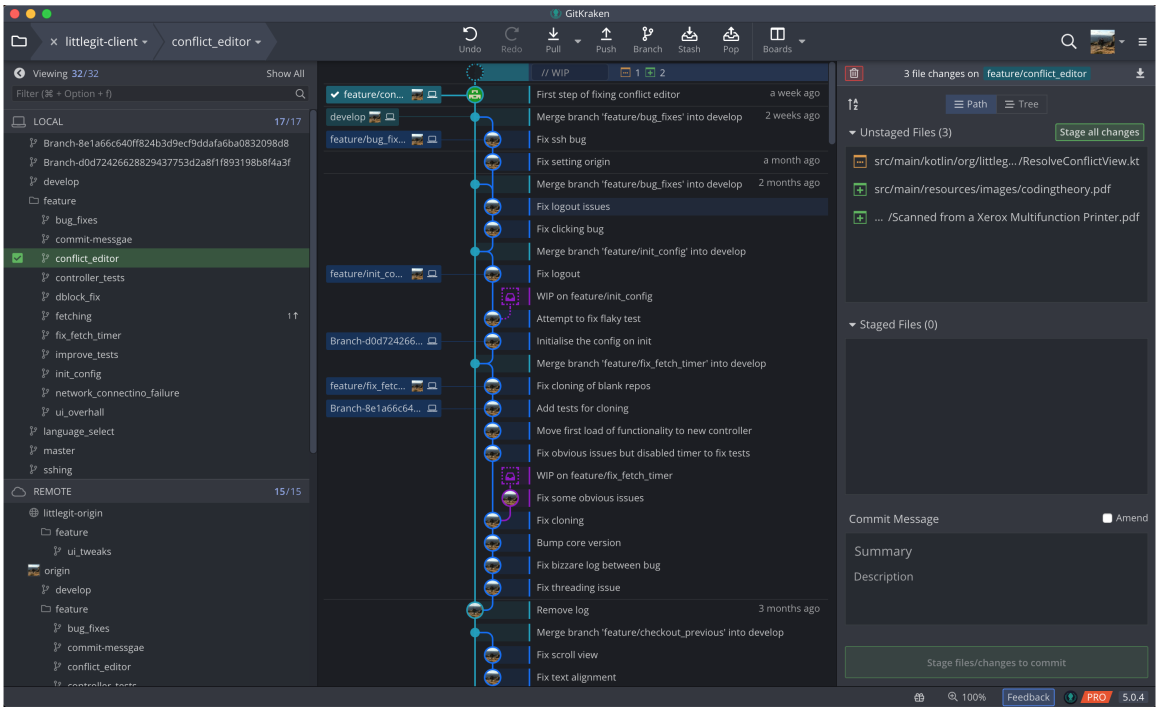
We discuss the GUI's design and features in the following section.

2.3 UI Design

Early in the project, the interfaces of other Git GUIs were examined, exploring their strengths, weaknesses and most importantly, their common traits. The findings of this research are discussed here as well as a discussion of how the results impacted the design of this project's user interface.

The three main applications researched are the application SourceTree [6] (mentioned earlier), GitKraken [7] and GitK [11]. Figures 2.3, 2.4 and 2.5 show a screenshots of their main interfaces.

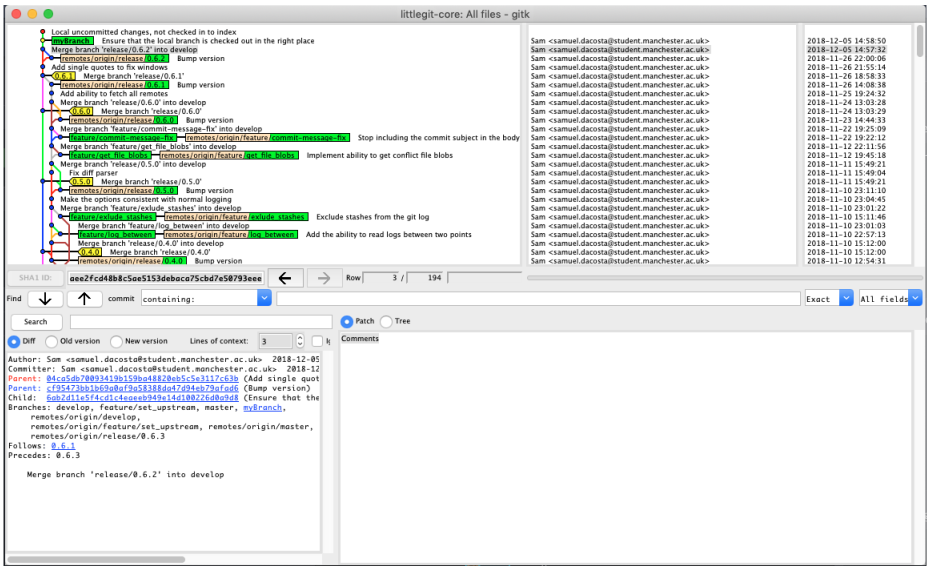
The first thing to note in looking at the interfaces is the clear common element being the Git graph. Though each has a different style and displays the information differently, the graphs are all large clear features of each interface. The decision was made to emulate the overall style in littlegit. The vertical graph, new commits on top with older commits in descending order below them. The reasoning here is that the Git history portrayed in this way conveys all needed information, but it is easy to present a simplified version (as was needed



```
First step of fixing conflict editor                      a week ago
Merge branch 'feature/bug_fixes' into develop             2 weeks ago
Fix ssh bug
Fix setting origin                                        a month ago
Merge branch 'feature/bug_fixes' into develop             2 months ago
Fix logout issues
Fix clicking bug
Merge branch 'feature/init_config' into develop
Fix logout
WIP on feature/init_config
Attempt to fix flaky test
Initialise the config on init
Merge branch 'feature/fix_fetch_timer' into develop
Fix cloning of blank repos
Add tests for cloning
Move first load of functionality to new controller
Fix obvious issues but disabled timer to fix tests
WIP on feature/fix_fetch_timer
Fix some obvious issues
Fix cloning
Bump core version
Fix bizzare log between bug
Fix threading issue
Remove log                                                3 months ago
Merge branch 'feature/checkout_previous' into develop
Fix scroll view
Fix text alignment
```

```
3 file changes on feature/conflict_editor
Unstaged Files (3)
src/main/kotlin/org/littleg.../ResolveConflictView.kt
src/main/resources/images/codingtheory.pdf
... /Scanned from a Xerox Multifunction Printer.pdf
Staged Files (0)
```

Figure 2.4: A screenshot of the main interface of the application GitKraken.



```
Local uncommitted changes, not checked in to index
myBranch  Ensure that the local branch is checked out in the right place      2018-12-05 14:58:50
Merge branch 'release/0.6.2' into develop                                      2018-12-05 14:57:32
remotes/origin/release/0.6.2  Bump version                                     2018-11-26 22:00:06
Add single quotes to fix windows                                               2018-11-26 21:55:14
0.6.1  Merge branch 'release/0.6.1'                                            2018-11-26 18:58:33
remotes/origin/release/0.6.1  Bump version                                     2018-11-26 14:08:38
Add ability to fetch all remotes                                               2018-11-25 19:24:32
Merge branch 'release/0.6.0' into develop                                      2018-11-24 13:03:28
0.6.0  Merge branch 'release/0.6.0'                                            2018-11-24 13:03:29
remotes/origin/release/0.6.0  Bump version                                     2018-11-23 14:44:33
Merge branch 'feature/commit-message-fix' into develop                         2018-11-22 19:25:09
feature/commit-message-fix  remotes/origin/feature/commit-message-fix  Stop including the commit subject in the body   2018-11-22 19:22:12
Merge branch 'feature/get_file_blobs' into develop                             2018-11-12 22:11:56
feature/get_file_blobs  remotes/origin/feature/get_file_blobs  Implement ability to get conflict file blobs   2018-11-12 19:45:18
Merge branch 'release/0.5.0' into develop                                      2018-11-11 15:49:21
Fix diff parser                                                                2018-11-11 15:49:04
0.5.0  Merge branch 'release/0.5.0'                                            2018-11-11 15:49:21
remotes/origin/release/0.5.0  Bump version                                     2018-11-10 23:11:10
Make the options consistent with normal logging                                2018-11-10 23:04:45
Merge branch 'feature/exlude_stashes' into develop                             2018-11-10 23:01:22
feature/exlude_stashes  remotes/origin/feature/exlude_stashes  Exclude stashes from the git log   2018-11-10 15:11:46
Merge branch 'feature/log_between' into develop                                2018-11-10 23:01:03
feature/log_between  remotes/origin/feature/log_between  Add the ability to read logs between two points   2018-11-10 22:57:13
Merge branch 'release/0.4.0' into develop                                      2018-11-10 15:12:00
0.4.0  Merge branch 'release/0.4.0'                                            2018-11-10 15:12:00
remotes/origin/release/0.4.0  Bump version                                     2018-11-10 12:54:31
```

```
SHA1 ID: aee2fcd48b8c5ae5153debaca75cbd7e50793eee    Row 3 / 194
```

```
Author: Sam <samuel.dacosta@student.manchester.ac.uk>  2018-12-05
Committer: Sam <samuel.dacosta@student.manchester.ac.uk>  2018-12
Parent: 04ca5db70093419b159ba48820eb5c5e3117c63b (Add single quot
Parent: cf95473bb1b69a0af9a58388da47d94eb79afad6 (Bump version)
Child:  6ab2d11e5f4cd1c4eaeeb949e14d100226d0a9d8 (Ensure that the
Branches: develop, feature/set_upstream, master, myBranch,
    remotes/origin/develop,
    remotes/origin/feature/set_upstream, remotes/origin/master,
    remotes/origin/release/0.6.3
Follows: 0.6.1
Precedes: 0.6.3

    Merge branch 'release/0.6.2' into develop
```

Figure 2.5: A screenshot of the main interface of the application GitK.

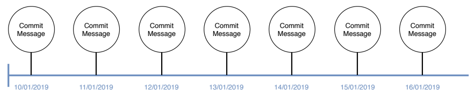


Figure 2.6: An early design for the Git graph using horizontal scrolling.

for littlegit) by simply omitting some of the information surrounding it.

Though other options were considered, Figure 2.6 shows an early design for an alternative graph. The idea was to emulate a timeline to help visualise the repository as being a series of snapshots of the code, allowing the user to navigate backwards and forwards along it. This idea however soon proved impractical, firstly visually representing branches in this model becomes messy and complicated, especially for more complex repositories. Furthermore, according to user testing [13], users dislike horizontal scrolling.

Returning to our discussion of the other applications, the graphs of both SourceTree and GitK are relatively cramped. A large amount of information in very close proximity, this is especially true of GitK. On the other hand, GitKraken takes much more space to display the same information, a feature it was important for littlegit to emulate to ensure the data is clear and easy to understand.

The next point to make is that all three applications have lots of buttons and menus. By looking through all the buttons and options, it became clear that in a system that aims to automate as much as possible. The vast majority of the options were unneeded. For example, SourceTree has three separate buttons for pushing, pulling and fetching. The question littlegit asks is why? If there is something to push to the server, why not just do it on the users' behalf, and if there are no new changes on the server to be pulled, then why present the user with the option? This lead to the decision of having a popup dialogue appear when the user's local version is behind the server, giving them the option to update without the need to clutter the primary interface with the option.

Since this automation is taking place, it seems prudent to explicitly describe the Git workflow littlegit aims to present. The workflow is built around the commit, creating commits should drive everything else. Creating a commit causes changes to be pushed to the server, and the arrival of new commits on the server should cause the application to update the repository accordingly. Branches in this workflow are much less significant and are only used by littlegit internally when needed. With this in mind, looking back at SourceTree especially, it is very non-obvious how to create a new commit, something imperative to avoid in littlegit.

Hence, creating new commits should be the primary operation the user sees when using the application. Furthermore, by looking at GitKraken and SourceTree in particular, the reader may also notice that many of the buttons on their left toolbars are merely the names of branches. In regular use clicking on them allows the user to check out these branches. Both applications also allow for using the graph to navigate between branches, allowing the user to click on the branch they wish to check out. With littlegit we aimed to expand this functionality, allowing the user to check out any commit in the history and handling this with branches internally without exposing this to the user.

2.4 Designing for teenagers

There has been much research conducted on the topic of designing user interfaces for teenagers. Unfortunately, the orientation of much of that research was towards websites and mobile applications rather than desktop apps. A result of this is usually much discussion of the importance of social media integration and good mobile compatibility, neither of which apply to a desktop application such as littlegit.

However, many of the conclusions are useful, particularly a study by the Neilsen Norman Group concerning designing for teenagers [12].

One of the biggest influencers of this study on the design of littlegit is the conclusion that teenagers hate childish content, seeing it as patronising. For this reason, littlegit refrains from any unneeded graphics and multimedia. Where an explanation of functions are required, they are short and do not talk down to users.

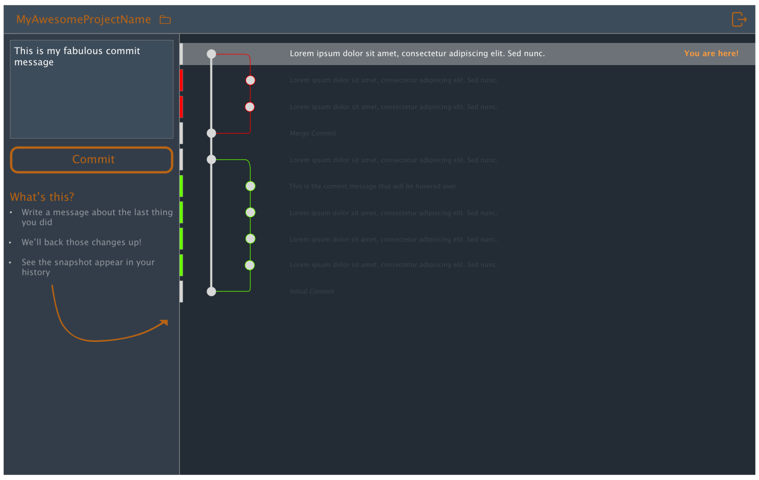
Furthermore, the study suggests that the strategy mentioned earlier of reducing clutter on the main screen is the correct approach, including refraining from using flashy animations and graphics.

Lastly, a point continually emphasised by the study is that of fast loading speeds. Though this refers specifically to websites, the study highlights the impatience of teenagers while using the software. The second objective of the project; therefore, of a scalable architecture which allows the application to work at speed is doubly important when considering the target audience.

2.5 Final design before implementation

The intent was to use an iterative and incremental approach to developing the GUI element of the software. However, having a limited opportunity to seek feedback from teenagers it was essential to have an initial design (influenced by the research above) to aim for and to take feedback into account when it was possible.

To do this the design software Sketch [**?**, 8] was used to produce initial designs. These designs are included in Figures 2.7 and **??**.


MyAwesomeProjectName

This is my fabulous commit message

Commit

What's this?

- Write a message about the last thing you did
- We'll back those changes up!
- See the snapshot appear in your history

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc. You are here!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Merge Commit

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

This is the commit message that will be hovered over

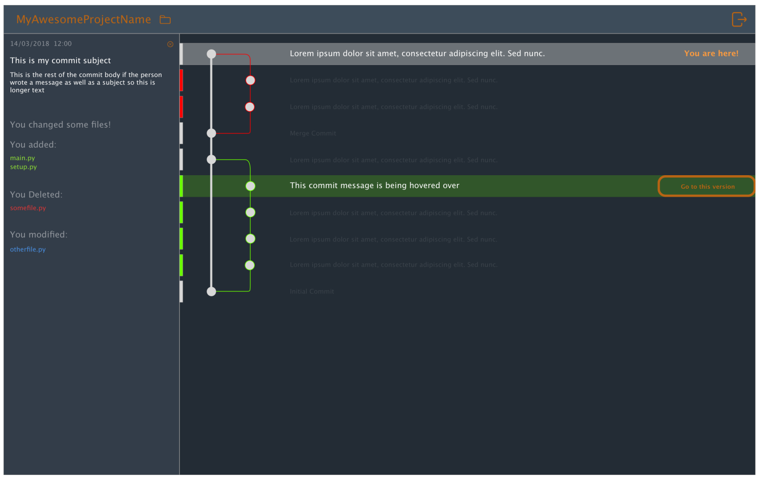
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Initial Commit

Figure 2.7: The main screen of littlegit


MyAwesomeProjectName

14/03/2018 12:00

This is my commit subject

This is the rest of the commit body if the person wrote a message as well as a subject so this is longer text

You changed some files!

You added:
main.py
setup.py

You Deleted:
somefile.py

You modified:
otherfile.py

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc. You are here!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Merge Commit

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

This commit message is being hovered over Go to this version

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nunc.

Initial Commit

Figure 2.8: The main screen of littlegit after a user clicks a commit from the graph.

Chapter 3

Implementation and Development

In this chapter, we discuss the main development phase of the project and justify some of the main decisions made throughout its course. We will explore the implementations of the architectures discussed at a high level in Chapter 2 as well as some of the more intriguing problems encountered throughout development.

3.1 Planning and management of work

Trello boards were used from the very beginning to keep track of project goals and tasks. A separate board kept track of tasks relating to the server, the littlegit-core and the GUI. Figure 3.1 shows the current state of the board for the littlegit-core (though the screenshot cuts off many "done" tasks). The strategy used with regards to the boards was as follows. Very early in the project, a large number of tasks were created in each board and placed in the *Backlog* column. Every week a few of the highest priority tasks were moved to the next column along, the *Next Priorty* column for completion that week.

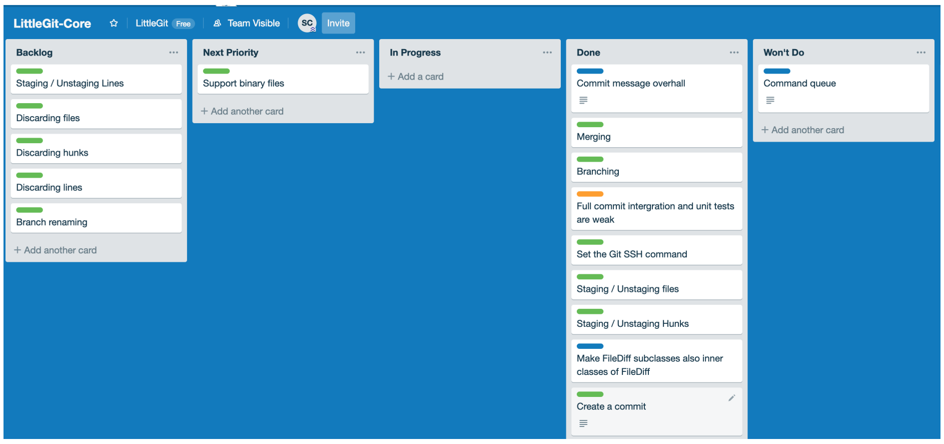
The strategy was first to complete a bulk of the tasks for the littlegit-core, then the server. Once both were stable, we then began work on the GUI. The approach worked reasonably well but did lead to a small amount of thoroughly tested code not being used in the final project; we will discuss the benefits and weaknesses of the approach in Section **??**.

It is worth mentioning the way tasks were chosen. The approach was to take the desired feature set (discussed back in Section **??**) and to break these features up into the tasks needed to achieve them.

3.2 Choice of language

In Chapter 2 we introduced the need for a library (the littlegit-core) to interact with the Git binary and provide a higher level interface to it, to be used by both the main server and the GUI.

Considering the time constraints on this project, only implementing the library once was crucial. Furthermore, the time constraints would not permit


LittleGit-Core ☆ LittleGit Free Team Visible SC Invite

Backlog
- Staging / Unstaging Lines
- Discarding files
- Discarding hunks
- Discarding lines
- Branch renaming
- + Add another card

Next Priority
- Support binary files
- + Add another card

In Progress
- + Add a card

Done
- Commit message overhaul
- Merging
- Branching
- Full commit intergration and unit tests are weak
- Set the Git SSH command
- Staging / Unstaging files
- Staging / Unstaging Hunks
- Make FileDiff subclasses also inner classes of FileDiff
- Create a commit

Won't Do
- Command queue
- + Add another card

Figure 3.1: A screenshot of the Trello board for the littlegit-core component of the project, taken after the end of development.

writing further wrappers of the littlegit-core to make it compatible with other technologies. Hence the GUI and server needed to be able to import the littlegit-core directly.

Three main technologies were considered. They are as follows.

1. **JavaScript:** Though this technology has been a staple of web development for many years, in recent years the Electron [1] has become a common technology allowing JavaScript, HTML and CSS (all traditionally web technologies) to be used to produce cross-platform desktop applications.

   Furthermore, NodeJS [3] is a technology allowing JavaScript's use in writing server code, making it a very viable choice as a core technology for littlegit.

   However, JavaScript is a *weakly typed* language, and previous experience has indicated that maintaining a large body of weakly typed code quickly becomes very difficult. Despite the advent of tools such as *TypeScript* which is a typed language which transpiles into JavaScript, it is still no replacement for a truly typed language such as Java.

2. **C# and .NET:** There exists rich support for developing web server applications using the ASP framework in C#, and the project very nearly was written entirely in C#, a modern feature rich language.

   However, early experimentation developing a toy C# application in an OSX environment (used for development) proved difficult and the tooling clumsy.

3. **Kotlin:** In the end, the project was written entirely in the Kotlin programming language. Kotlin is a strongly typed language that can compile to the JVM and is fully compatible with Java frameworks and libraries enabling the use of longstanding, reliable frameworks to be used (as we will discuss in subsequent sections).

Kotlin is a relatively new language, and though it is inheritance model and general structure is similar to Java (to allow compatibility), it is a very different language to program in. Its rich support for functional programming was heavily utilised primarily in the development of the littlegit-core in the parsing of raw git output. Furthermore, Kotlin is far less verbose than Java allowing much faster development with less boilerplate code to maintain.

3.3 The littlegit-core

As has already been discussed, this is the library that interacts directly with the Git binary. The primary challenge involved in producing this component was the interaction with the so-called git-plumbing.

Git commands are loosely grouped into two categories (though the distinction is not a hard one). The first is git-porcelain. Porcelain commands are those with which most Git users are likely to have interacted. A good example is `git-status`, which gives a summary of the current state of the user's local Git repository. These are commands designed for use by human users; often, their outputs are tweaked between Git versions to aid human readability. This makes them unsuited for machine use; it is impossible to reliably parse output which is slightly different in every version of Git.

The git-plumbing commands, on the other hand, are generally lower level commands to the porcelain and are often used under the hood by the porcelain itself. Their outputs are considered stable and designed for interaction with other software rather than humans. While making them a far better choice for the littlegit-core to interact with, the difficulty is in the fact they are by nature lower level commands. For example, to check out a branch with git-porcelain is a straightforward one-line operation, but this conceals multiple plumbing commands under the hood. Littlegit-core endeavours to use plumbing commands wherever possible.

As has been mentioned, the project aims to follow best practices wherever possible. For this reason, and the fact that the rest of the project relies heavily on the littlegit-core being stable, a TDD testing approach was used. One way of achieving this was to *design the library for testibility*. The main design decision supporting this was the decision to make the library's operations synchronous. The user's interaction with the GUI should not grind to a halt because a shell command is executing. Hence there must be some asynchronicity, but the decision was made that the GUI would be responsible for threading calls to the littlegit-core, leaving it to return its results synchronously.

Methods who return synchronously prove to be a much simpler and more reliable to test, allowing the entire library to be covered thoroughly with tests.

3.3.1 Architecture

The primary driver behind the architecture of the littlegit-core is the fact it must be able to execute its commands on remote machines over an SSH connection as well as on the local machine. For example, it is used to initialise remote repositories via SSH on the Git servers.

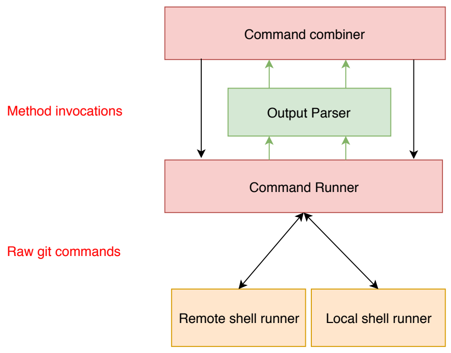


Figure 3.2: The internal architecture of the littlegit-core.

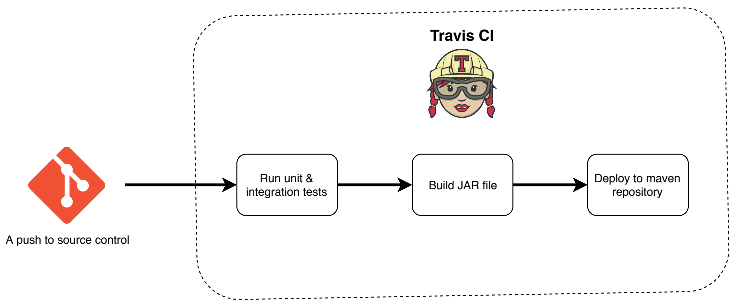


Figure 3.3: The deployment pipeline of the littlegit-core.

Figure 3.2 shows the way the library is architected. The top layer is used to combine plumbing commands; for example, multiple plumbing commands are required to change branches. It makes calls to the `command runner` which provides a high-level interface to these Git commands. The `command runner` then converts these high-level invocations raw commands to be interpreted by Git, which are executed either through the local or remote runners depending on the configuration. Local for the GUI, remote on the server.

3.3.2 Deployment

We keep mentioning that the library is *used* or *included* by the GUI and server. It is worth explaining what this means and describing the deployment process for the library. Figure 3.3 shows the deployment pipeline, triggered by a push to source control; all the automated tests are run remotely by Travis's continuous integration service. Importantly when the tests succeed the library is deployed to a maven repository allowing it to easily be included as a maven dependency by the server and the GUI, without the need to manually copy any files.

3.4 The main server

In this section, we will explore the implementation of the main server and justify the technologies used. Recall that the main server's primary function is the management of users and remote repositories. Keeping track of user details as well as information on which users have access to which repositories.

3.4.1 Technologies

We have already justified the decision to use the Kotlin programming language for the server. However, multiple options were available in regards to different technologies for everything from the database to the framework used to build the server itself. We discuss and justify the choices made here.

Firstly, Amazon Web Services hosts all the online infrastructure for this project. The justification for this is simple, hosting of databases and servers is expensive, and AWS offers a free tier of the services needed for students.

Web server framework

Let us first discuss the choice of the server framework used. The two frameworks most seriously considered are as follows.

- The Spring framework [15].
- The Jersey framework [2].

We chose Jersey for its lightweight nature, it provides an easy way of creating API endpoints but unlike Spring does not attempt to control the entire stack down to the database allowing much more flexibility. Considering the need for the server to communicate via SSH with the Git servers, having full control of the stack was vital.

The downside of Jersey and the advantage of Spring is a direct consequence. Spring provides much of the required functionality (communication with the database and authentication of users, being most significant) as standard, while these must be done manually with Jersey. In the end, the need for flexibility made Jersey the correct choice, despite the additional implementation work involved.

Database and caching

Given the rise of non-relational databases such as CouchDB and Firebase the decision to use a traditional SQL database is no longer a trivial one. These non-relational databases have some significant advantages, one of the most enticing being their quality of scaling horizontally, unlike traditional databases which generally only scale vertically.

However, the decision was made to use a traditional database for the following reasons. The first and most important of these reasons are the need for complex querying on the data where non-relational databases do not compete with the power of traditional SQL. An example of this kind of complex query is the need to find all the Git servers which contain repositories the user either owns or has access to.

Moreover, the data has an evident structure, a structure which does not need to change often or adapt to a rapidly changing environment (where non-relational databases come into their own). Because of this, the static, rarely changing schemes of traditional SQL databases are not a hindrance.

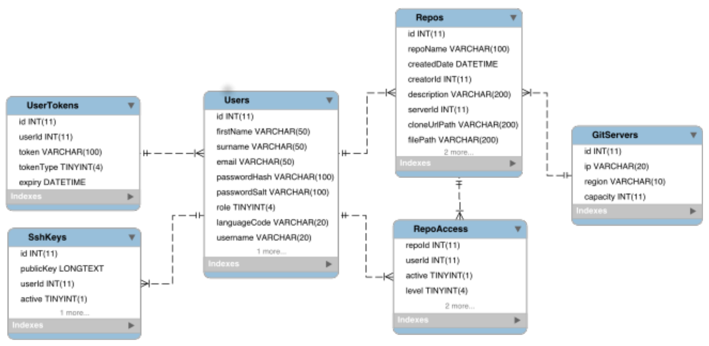


Figure 3.4: An entity relationship diagram representing the database.

The scaling issue, however, must be addressed. No matter the convenience to the programmer, the use of a technology that will not scale efficiently cannot be justified when one of the objectives of the project is to produce a scalable system.

Let us consider the data that the system needs to store. For the reader with experience with relational databases, Figure 3.4 shows an entity relationship diagram for the database. In short, the data we need to store is as follows.

- User information including tokens used for authorisation and authentication.
- Information about repositories and the servers upon which they are stored.
- The SSH keys for each user.
- Information regarding which users have access to which repositories.

The scalability of the server relies on one key observation, the fact that the reading from the database is a far more common operation than writing. For example, the creation of repositories is not a common operation but checking if a user has access to a repository is very common (this will be discussed further in subsequent sections). This observation allows us to introduce caching to solve our scalability issue. By caching as many of the values as possible in a scalable, **in-memory** caching solution, we massively reduce traffic to the database. We will discuss this further in subsequent sections, for now, we will justify the use of Redis [5] as our caching solution.

Redis's feature set is vast [5], but the important ones for us are its in-memory key-value store and built-in replication. Its in-memory nature allows for very fast reads of the data. Of course, key-value pairs do not offer the benefit of flexible SQL-like queries. However, this isn't what we need at this point, if we need to do complex queries we can access the database, the cache speeds up common simple operations such as returning the details for a particular user, or checking if an authentication token is valid **without** the overhead of checking the database which is on disk.

Moreover, having a cache with built-in replication allows it to scale across multiple machines easily giving us the horizontal scaling we require. The reader may wonder why we are using a caching solution rather than making use of data

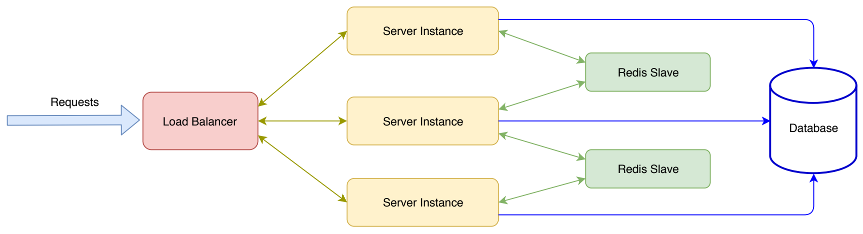


Figure 3.5: The overall architecture of the main server and the way it is designed to scale.

structures within the server code itself to achieve the same thing. It is a very valid argument to say that such an approach would be more straightforward and easy to implement, with less to go wrong!

Let us consider the situation where the load on the server becomes large, causing requests to become slow and even to time out. Since the memory of the machine hosting the server caches the data, the only option is to get a bigger machine with more memory (vertical scaling). On the other hand, using our approach we can spin up multiple machines hosting the server to balance the load, and Redis's inbuilt replication allows us to do the same thing with the cache. Figure 3.5 shows the idea [1].

Data transfer

The decision to use the JSON data format for communication between the server and desktop client was a straightforward one. JSON is lightweight and is easily parsable by machine while also being straightforward to understand by humans (making debugging easy) [9].

Furthermore, the decision to create a RESTful interface with the server allows it to be used by other clients in future easily. For example, if a web interface was required to enable users to view and manipulate their remote repositories, the same endpoints could be used without any additional work.

3.4.2 Internal server architecture

While discussing technologies, we saw the overall structure of the server technology stack. In this section, however, we discuss the way the code which interacts with these various components is architected.

As was described in Section 3.4.1, a lightweight server framework was selected for the system which does not enforce a particular architecture. The Spring framework was a loose inspiration to our architecture, resulting in the use of a *Layered Architecture*. Figure 3.6 shows the different layers of the system.

As the diagram shows, each layer abstracts the one beneath it, and each layer interacts exclusively with the one below it. The roles of the individual layers are as follows.

[1]This was only tested with one machine each for the cache, server and database without a load balancer due to the cost of doing so on Amazon Web Services and other platforms, the code however fully supports the architecture described.

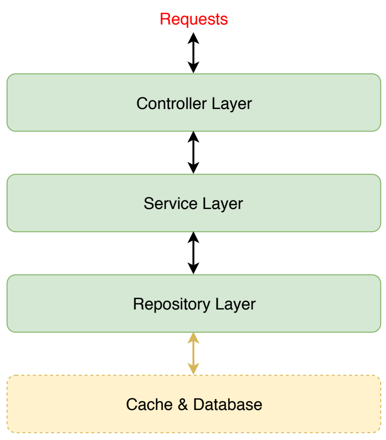


Figure 3.6: The internal architecture of the main server.

Repository Layer

The repository layer is little more than an abstraction on the database and caching layers. Used by the service layer to retrieve and write information, this layer invokes raw SQL commands to communicate with the database and crucially also reads from the cache, and invalidates cached information when data is updated.

Furthermore, the repository layer handles the mapping of data from the database and cache to Kotlin data structures.

By the nature of this layer's direct communication with the SQL database, it is here that considerations must be made to protect against SQL injection attacks.

Service Layer

In this layer, we have all of our application-specific business logic. For example, the process of creating a new repository consists of several steps, involving multiple writes to the database and the running of commands on remote Git servers. The logic for combining these operations exists in the service layer.

Controller Layer

This layer defines the interface between the server's logic and the outside world. This means defining the individual API endpoints and the mappings from HTTP URLs to methods. Importantly, in this layer, we also define what kind of user can access each endpoint. Admin users have access to everything which we mainly use for testing; then we have standard users of the system. The third type of user is the unauthenticated user; we can think of these users as those that are not logged in. Of course, the only endpoints these have access to are those used to login and register new users.

By far, this is the most difficult layer to test since it how we define the interface of our API using syntax provided by the Jersey framework. For this reason, it is vital that the controller contains as little business logic as possible.

In fact, for this project, each controller does nothing but invoke code in the Service layer which is far easier to test.

3.5 Desktop GUI

Some of the work involved with building the GUI consisted of standard implementation of user interfaces. Placement of buttons, wiring up of forms e.t.c. In this section, we skate over these tedious parts and also omit discussion of functionality implemented in the GUI already mentioned in other sections.

3.5.1 Git graph

One of the only elements of this project involving the design of an algorithm is the drawing of the so-called *Git graph*, which is a visual representation of the project's history.

Some readers may suppose that given the feature set described in Section **??**, the resulting history will not be particularly complex since the user is unable to create and merge branches. However, we must consider the situation where an existing (complex) Git repository is opened in littlegit; it is essential that the graph displays correctly. For this reason, the software supports arbitrarily complex Git graphs. Figure 3.7 shows a project with a reasonably complex history opened in littlegit.

Each time the Git history changes (for example when the user creates a commit) the local copy needs to be updated, the graph must be re-drawn. As we saw from our research in Section 2.4 it is critical that this is as fast as possible.

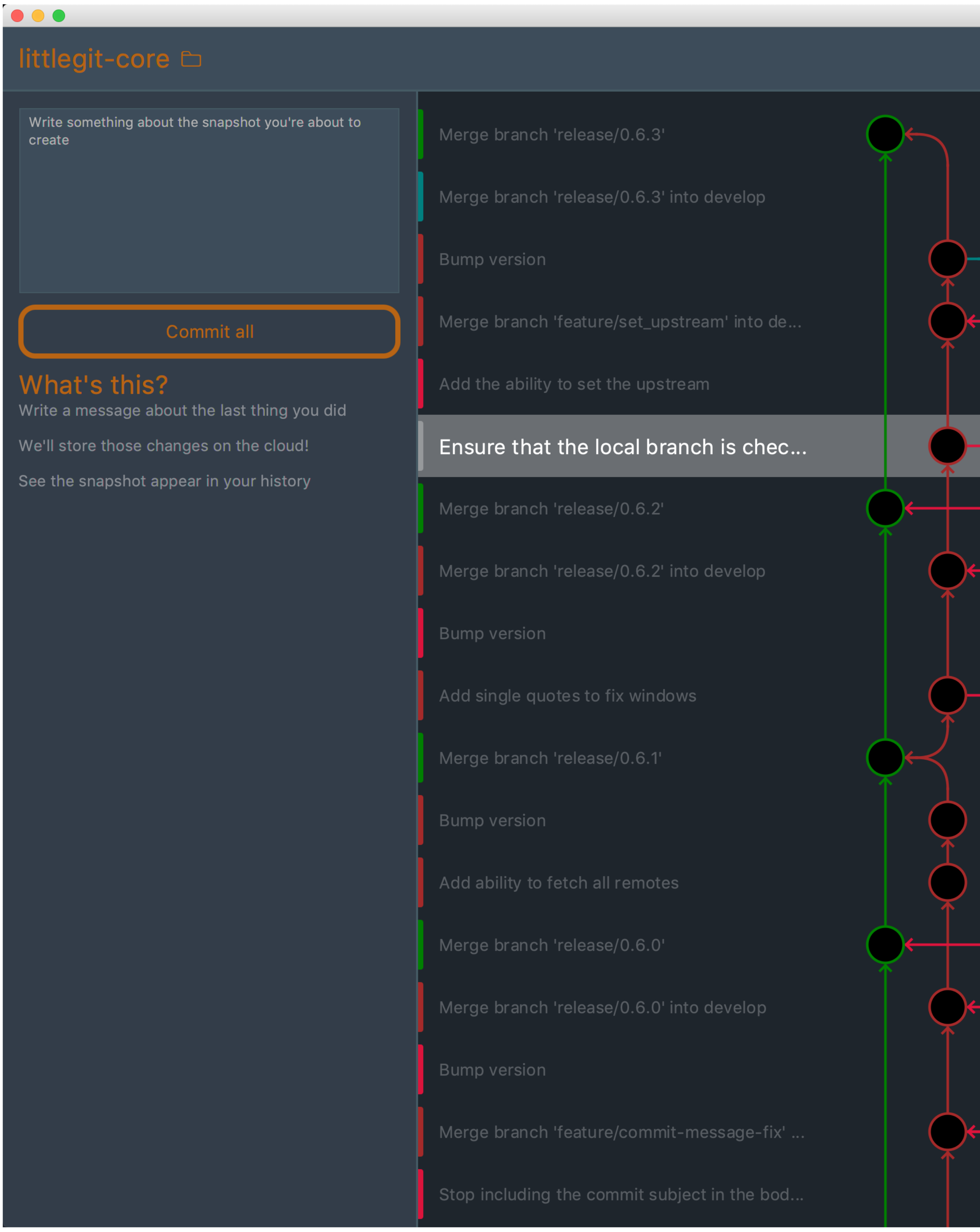
Consider a project with a long history with many thousands of commits. It is not unreasonable that a person from the target demographic may be working with such a project; for example, it may be an exercise to make changes to an open source piece of software. In this case, the graph must update quickly. The key to this is first to calculate which part of the graph is currently visible on the screen based on how far the user has scrolled and the size of the window.

Experimentation with the resulting algorithm has shown that the time to redraw the graph (which is now independent of how extensive the history is) is negligible in comparison with the time taken to read the history from Git in order to draw it.

3.5.2 Threading

We have stated repeatedly the importance of software that does not lag considering the target audience. It follows directly that the software should not freeze just because it is performing background tasks. In practice, this means ensuring we do not perform long running tasks on the application's main UI thread.

Long-running tasks include API calls (since the network may be slow), interactions with the file-system, including calls to the littlegit-core since invoking some Git commands may be slow. On the face of it, this seems straightforward. Upon needing to run one of these tasks, we run it on a separate thread and wait for it to complete before updating the UI.


littlegit-core

Write something about the snapshot you're about to create

What's this?

Write a message about the last thing you did

We'll store those changes on the cloud!

See the snapshot appear in your history

Merge branch 'release/0.6.3'

Merge branch 'release/0.6.3' into develop

Bump version

Merge branch 'feature/set_upstream' into de...

Add the ability to set the upstream

Ensure that the local branch is chec...

Merge branch 'release/0.6.2'

Merge branch 'release/0.6.2' into develop

Bump version

Add single quotes to fix windows

Merge branch 'release/0.6.1'

Bump version

Add ability to fetch all remotes

Merge branch 'release/0.6.0'

Merge branch 'release/0.6.0' into develop

Bump version

Merge branch 'feature/commit-message-fix' ...

Stop including the commit subject in the bod...

Figure 3.7: The Git graph for the littlegit-core library opened in littlegit.

Unfortunately, the fact that we are interacting with a Git repository complicates matters. Consider the scenario we make a call to the littlegit-core to create a new commit on a very slow machine, suppose that multiple files have changed and must be included in the commit. Now suppose the user tries to check out a different commit. Checking out a different commit while halfway through the creation of another will lead to Git throwing errors and possibly corrupting the repository (depending on precisely what happened). To prevent this, we have two options; whenever any background task runs, we disable the entire UI to prevent the user from doing anything. As we have discussed this is bound to anger our users.

The second option, the option used by littlegit is to use a queue instead. All operations which depend on these long-running tasks are added to a queue and executed one by one (on a separate thread). The scenario described above now performs as a user would expect, one operation following the other automatically without the need for repeated clicking.

3.6 Authentication

There are two main kinds of authentication used by the system. SSH authentication is used when the desktop GUI performs Git operations interacting with the remote Git server (for example `git push` and `git fetch`), whereas calls to the API use a token-based system.

3.6.1 API authentication

The token-based system used to authenticate calls to the API implements the OAuth 2.0 protocol since at the time of writing this is the industry standard for authentication [4]. Furthermore, it is relatively simple to implement, and the use of tokens also makes it easy to cache authentication details for a user which helps speed up requests.

3.6.2 SSH authentication

Implementing secure SSH based authentication between the desktop GUI and the Git servers is one of the key technical challenges of the project. Firstly we must explore what happens when we run a command such as `git fetch`. Internally, Git runs a command using SSH on a remote machine. The context under which the command has a user in the environment of the remote machine, by convention this user is called *git*.

The first task was to ensure that the user *git* could only execute Git commands since otherwise, it would be possible to execute any shell command on the remote machine, a clear security risk. To achieve this, the *git* user is configured not to use the standard Unix shell, but instead, to use a tool called the **git-shell** which performs the restrictions (this is reliable since the creators of Git itself wrote the tool).

On the other hand, when a user creates a new repository, it is the responsibility of the main server to create this repository on one of the Git servers. This requires a different user to the *git* user since it needs permission to create a directory and initialise the repository. Due to this need, it was important to carefully set file permissions to ensure both users have only the permissions they need and no more to reduce the threat of malicious access.

To explain the way the actual authentication process works we will work through the steps of an example user action. In the following example, the user is creating their first repository and then pushing to it. Note, when we refer to API calls in this section we still refer to calls to the server's RESTful interface using token-based authentication.

Step 1: Before creating the repository

The act of logging in to the desktop GUI for the first time on a machine triggers the generation of the user's public/private SSH key pair. An API call registers the public SSH key with the main server. SSH. If the user already has repositories (for example one created from a different machine), then the main server produces a list of all the Git servers hosting repositories owned by that user. It then takes the public SSH key and adds it to the server's `authorized_keys` file, which contains a list of all the public SSH keys whose users have permission to access that machine.

Step 2: Creating a repository

It is the desktop GUI's responsibility to initialise the Git repository locally and to make a call to the main server to set up the remote repository which it does by executing commands via SSH on one of the Git servers (both making use of the littlegit-core). The GUI must also set a Git configuration option to ensure Git uses the generated SSH key for authentication and not look in the location[2]. This prevents interference with user-created SSH keys on machines used by more advanced users.

Step 3: Pushing or fetching from the remote server

When the GUI triggers a fetch or pull from the server, the first thing SSH does is looks up the user's public SSH key in the list in the `authorized_keys` file. An example entry in this file is as follows.

```
command="./check-authorization.sh 677",no-port-forwarding, ...
no-x11-forwarding,no-agent-forwarding,no-pty ...
ssh-rsa AAAAB3NAAADAQABAAAAgQDmFNE9u8MHeGhb2o6qCfv8eRjK18EUA75XH3vElQ== SSHCerts
```

In general, an entry in the file has the following format.

```
command="./check-authorization.sh <user ID>" <options> <user's ssh key>
```

Once SSH finds the user's ssh key in the list (if it is not present the request is immediately terminated), the *command* specified on the line is executed with the user's ID as an argument. The command has access to information about the Git command being executed, including the repository the request is attempting to manipulate. Using this information the Git server can call an endpoint on the main server to check if the user with the given ID has access to the given repository. If so, the request is allowed, SSH terminates the request.

[2]This is `~/.ssh/id_rsa` on most Linux systems

Unfortunately, this presents us with a potential performance bottleneck. The list is searched sequentially for the user's SSH key. If the list is extensive, this search quickly becomes slow which is further motivation to use multiple Git servers since having multiple machines hosting the repositories, each with only a subset of the total SSH keys registered in their `authorized_keys` files leads to far fewer lines in each file.

Chapter 4

Evaluation and Testing

This Chapter discusses the approach to testing and evaluating the system; this includes both the automated testing strategy and the steps taken to procure feedback from likely users.

The focus of the automated tests was to ensure a stable and reliable product, to catch bugs and help prevent regressions. On the other hand, the aim of user testing was not on finding bugs (though this is a secondary objective) due to the limited amount of time available interacting with likely users (i.e. teenagers). It was decided that a much better use of this time was to obtain feedback about the usability and usefulness of the system, rather than spend the time having users try to break the system to discover faults.

4.1 Automated Testing

As has already been indicated, this project consists of three distinct codebases. As a result, the testing strategy for each is slightly different, and we will discuss them separately.

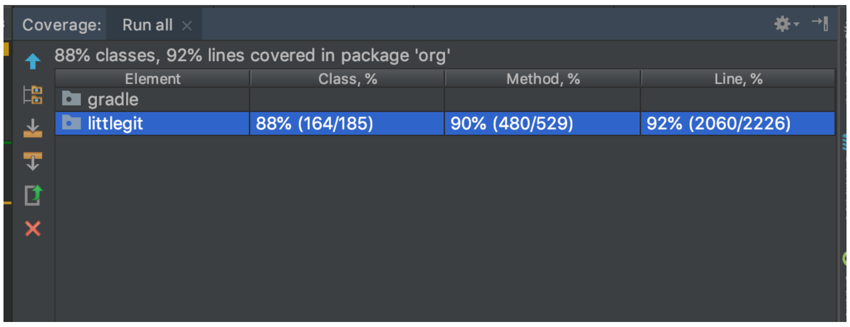
4.1.1 Littlegit-core

Conventional wisdom with regards to automated testing is to follow the **Cohn test automation pyramid** [16] which requires a large proportion of the tests to be unit tests and fewer service level and UI tests.

For testing the Littlegit-core, this guideline was purposely ignored, and the vast majority of the tests are integration tests. These examine the library's interaction with real Git repositories (created and managed by the test case) and ensure results are correct.

To follow the guidelines these tests would have to become Unit tests. To achieve this, we would have to mock Git's output at each stage of the test. However, this quickly proved impractical since minor changes to the test code require new dummy Git output to be generated each time, a slow manual process, resulting in the decision to test against Git itself.

The downside to our chosen approach is that test cases execute marginally more slowly due to the dependency on Git and the filesystem, though in practice



88% classes, 92% lines covered in package 'org'

Element	Class, %	Method, %	Line, %
gradle			
littlegit	88% (164/185)	90% (480/529)	92% (2060/2226)

Figure 4.1: The code coverage statistics for the littlegit-core library.

the suite of over a hundred tests still runs in under 15 seconds on average making this a non-issue for development.

Regarding code coverage, as Figure 4.1 shows we have code coverage above 90% which we consider more than acceptable. Furthermore, in using the library in the development of the project's other components, it has proven stable, with only a few bugs, implying the testing strategy has been successful.

4.1.2 The main server

The main server called for an entirely different testing strategy to that used for the littlegit-core. The chosen approach follows the conventional wisdom; many unit tests, fewer service level tests (of course no GUI tests since it has no GUI).

Recall the server architecture we discussed in Section 3.4.2. Many Unit tests are used to test the *Repository* layer, the layer that interacts with the database. The decision was made to test against a real database and cache rather than try to mock out the connections since this is tedious. We also get the advantage of checking the database and cache connections using this approach.

To Unit test the *Service* layer we mock out the dependency on the *Repository* layer to ensure we only test the Service layer functionality.

A much smaller number of integration tests are used to check the stack from the *Controller* level down to the database, the aim being to catch any issues in the integration of the various layers.

Formal Performance tests were not used in the development of the server. Manual testing shows that calls to the API execute quickly and that the caching strategy does indeed speed up data retrieval. Any further insights that formal Performance tests would provide would require the system to run on multiple machines as previously described leading to a significant monetary cost which it was deemed unnecessary.

Code coverage for the server is lower than the littlegit-core but still stands at 76% which is acceptable.

Improvements

Manual testing was used to sanity check the API endpoints. Running the server, and then calling the endpoints with input data and reviewing the JSON response. Automating this process would be useful. Currently, all tests are run automatically on a build server before deployment of the server. These planned tests would run after the server has been deployed, sanity checking that all the endpoints are working and that there are no glaring issues.

4.1.3 Desktop GUI

By far this codebase was the most difficult to test owing to the fact most of the operations most vital to examine are asynchronous. It was decided that UI tests would not be used since the Tornado FX framework's support for them is limited, and early experimentation showed them to be unreliable.

The strategy, therefore, was to concentrate on the code that performs automation of Git, ensuring that the system reacts correctly to Git errors and the user mistakes causing them. Furthermore, it was essential to verify the GUI responded correctly to issues with the network connection and errors from the API. Mocking was used to simulate these conditions and ensure the system reacted appropriately.

Unit testing was used extensively to ensure that the Desktop GUI's internal data persistence store was stable, tests that proved useful since minor mistakes in the persistence code almost caused multiple serious regressions which were caught by the tests.

Due to the decision not to use UI tests, the test coverage of the GUI is by far the lowest of the components at 33%. However, if we omit UI code from this figure, our coverage is close to 80% which we regard as acceptable.

4.2 User testing

As was briefly mentioned in Chapter 2, a true iterative approach to developing the GUI was not possible due to a limited number of opportunities to seek feedback from the target audience.

However, two feedback sessions occurred during development, the first roughly halfway through and the second nearing the end. The students involved had received a short talk about source-control and the idea of creating commits.

The format of both sessions was the same; participants were asked to perform tasks using the software (for example creating a few commits, then navigating back to a specific commit and examining it). As they completed the tasks, it was observed that none had difficulty using the software and upon being asked, all said they found it easy and intuitive. If anything some seemed confused why they were being asked to complete such trivial exercises. Of course, two focus groups is not enough to conclude the user experience is well designed and useful, but it is a promising indication.

Interestingly, the second group pointed out more issues with the software than the first. These issues were not bugs, but suggestions for improvement such as the opinion that the Git graph looks strange and its meaning is unclear when only one commit exists in the repository. Unfortunately, time constraints prevented their concerns from being adequately addressed during the main project lifecycle, but these are indeed areas for future improvement, making the feedback valuable.

Moreover, feedback sessions were carried out numerous times within a small tutor group involving students and lecturers where feedback was obtained about the project's progress. These sessions were instrumental in the initial designs described in Chapter 2, and certain features (the Git graph chief among them) were iterated upon multiple times based on feedback from these sessions.

Chapter 5

Reflection and Conclusion

As a learning experience, overall the project was successful and gratifying in that the end product is a piece of well tested, functional and useful software. In this Chapter, we briefly discuss what went well, and what could be improved upon concerning the software's development.

5.1 Management of work

The management strategy involving the use of Trello boards was overall successful. Work was done consistently throughout the allotted timeframe. Milestones were adjusted as work continued, but no significant changes were made.

In hindsight, these milestones should have been slightly tighter throughout the year to allow time to address feedback from the second focus group session.

Furthermore, as discussed in Chapter 2, it was decided to complete the littlegit-core first, followed by the server and GUI, the thought being that this would make the development of the latter two easier. However, this resulted in a small amount of code written in the littlegit-core that is never used by the other components. By employing the concept of *thin, end to end slices of functionality* as is preferred by Agile methodologies we could have avoided this.

5.2 Project goals

All the goals discussed in Chapter 1 were achieved, the desired functionality is in place, and feedback from users suggests it is useable and fit for purpose. In short, we are content with the overall results of the project and consider it a success.

5.3 What went wrong?

Of course, there were areas in which things could have been improved. We have already mentioned a few minor changes that could have improved the project's flow. The most significant change, however, if this project were to be started again would be the technology stack. Kotlin as a language indeed was a good choice. However, the use of the Tornado FX framework proved to be a mistake.

The problem lies in the fact it is built upon JavaFX which unfortunately the developer community seems to have rejected.

Support for the frameworks online is minimal, making development more difficult and more importantly making the software tough to distribute and install. In hindsight, despite the massive disadvantages of JavaScript as a language, use of the Electron framework which we disregarded in Chapter 3, may have resulted in better overall user experience.

5.4 Personal Achievements

I learnt a great deal in completing this project. Learning the Kotlin language including its functional aspects was certainly a valuable experience. This includes using it in writing a library learning to deploy it using Maven.

Though University did teach the basics of using Git, this project required digging much deeper into low-level Git commands and architecture. As a result, I believe myself considerably more knowledgable about Git, and its quirks than I was beforehand.

Scalability is also an issue I have never before had to consider in writing software at University. It was very new to me to try to write code designed to be run simultaneously on multiple machines and am very happy with the results! On a related note, security and authentication (especially using OAuth 2) is something I have never had to consider during my University career. Through this project, I learned about securing endpoints with token-based authentication, and of course about using more advanced features of SSH to secure the Git repositories themselves.

Lastly, I gained valuable experience in running focus group sessions, something I had never done before to obtain feedback from likely users. These sessions were incredibly valuable and indeed a form of obtaining feedback I will use in future!

Appendix A

Libraries

Chapter 3 details many of the pivotal decisions regarding technologies and libraries made throughout the project. However, there are a few libraries used that warrant credit, though the decision to use them followed directly from the decisions discussed in Chapter 3, or from merely being industry standard libraries for their given functions.

Retrofit [?] was used for performing API calls from the desktop GUI to the main server, and Moshi [?] used for the conversions between JSON to Kotlin objects.

For communication between the main server and the Redis cache, Jedis [?] was used, chosen for its speed and simplicity.

Dependency injection was used in architecting the main server code, improving readability and maintainability. This technique makes the architecture described in Chapter 3 feasible. The dependency injection framework chosen was Dagger 2 [?].

Lastly, though not a software library, the library of icons used for many of the buttons in the GUI was obtained under from a FlatIcon author [?] under the Creative Commons licence.

Bibliography

[1] Electron.

[2] Jersey.

[3] Nodejs.

[4] Oauth 2.0.

[5] Redis.

[6] Atlassian.

[7] axosoft.

[8] Bohemian Coding. Sketch.

[9] ereger. What is json?

[10] git scm. Git in other environments - graphical interfaces. Accessed: 26-02-2019.

[11] gitk.

[12] Alita Joyce and Jakob Nielsen. Teenagers ux: Designing for teens.

[13] Jakob Nielsen. Scrolling and scrollbars.

[14] Ofqual. Gcse outcomes in england. Accessed: 26-02-2019.

[15] Pivotal. The spring framework.

[16] Ham Vocke. The practical test pyramid.