Making Git Accessible For Children

Third Year Project

Samuel Da Costa

BSc (Hons) in Computer Science and
Mathematics
Supervisor: Caroline Jay

The University of Manchester

Abstract

This report discusses a project aimed to bring version-control to children through a simplified interface to the Git version-control system. An argument that such software is needed will be presented, and the design and implementation of the software discussed. This will include a discussion of how the system was split into distinct components and the individual challenges involved in implementing these components. We will discuss the testing strategy, both in terms of automated testing and the use of focus groups to obtain real user feedback. Finally, this report reflects on the achievements of the project and comments on decisions that would be made differently if the project restarted.

Acknowledgements

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. This is according to official statistics published by Ofqual [1]. Moreover, the UK government has introduced simple programming to children as young as five years old [2], 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 version-control system. However, students studying their GCSEs and A-Levels in Computer Science often produce non-trivial amounts of code, the need for complete backups and version-control is apparent.

In the field of computer science, this is a solved problem. The *Git* [3] version-control system 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. Using it effectively requires practice and an understanding of its low-level commands which can be daunting to new users.

This project aims to strike a balance between the complexity of Git and the needs of younger users. To take advantage of such a feature-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 such as, "rolling back," and, "saving a new version".

1.1 Background on Git

This section briefly describes some of the main advantages of Git and introduces 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 was 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 secure storage of the objects on a remote server (referred to as *pushing* to the remote server). Sharing of projects across devices now becomes simple. 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.

In practice, however, the use of these commands can often be fiddly for new users. 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, the result is usually an error which is very difficult to read and interpret. The project aims to automate the resolving of these issues, as well as performing actions automatically to prevent them.

1.2   Project Goals

At the outset of the project, there were three main objectives, from which all other aims extend. These are as follows.

1.2.1   Objective 1

The first objective was 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.

To define specific use-cases that the GUI should support, it was essential to consider the behaviour of Git users. A questionnaire was circulated to adult users who already use Git to determine which operations are considered most useful, and which are most challenging to use.

For the project, we aimed to support the most useful operations. For those useful yet challenging tasks, the aim was to support them but possibly in a simplified form. A prime example of this is branching, 96% of respondents indicated branching was important to their workflow, but only 66% found branching to be an easy task. Therefore, a simplified branching procedure is used whereby branches are automatically created and managed as the user navigates back and forth through the commit history.

Furthermore, the survey indicated that only 7.4% considered the setting up of SSH keys to be easy, with a further 77% saying they always needed to look

up instructions for how to do so. Therefore this project aims to handle all SSH authentication automatically.

The supported use-cases were also heavily influenced by studying the national curriculum for Computing in England for the target age group [2]. It reveals that students are not required to work in teams or work on the same code. Because of this, the GUI concentrated on a single user's needs, for example, the branching structure described above is ill-suited to a repository with multiple collaborators but provides a much simpler user experience for the single user [1].

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 to 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

1.2.2 Objective 2

The second objective was less tangible; 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. 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.

1.2.3 Objective 3

Finally, the project's third objective was to introduce the software to children and gather feedback about how useful it would be to them. This objective is fundamental; without user feedback and appraisal there is no way of knowing if the software is fit for purpose.

By the end of the project, two feedback sessions were conducted with school children during a Computer Science lesson at their school. We will discuss their feedback and the approach to the sessions in Section 4.2.

---

[1]The system does fully support multiple users interacting with a repository. However, we optimise the GUI for one user, possibly on multiple machines since this is the most likely scenario.

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 the rest of this report.

2.1   Other work in this area

So-called *Git GUIs* (graphical user interfaces to Git) have existed for almost as long as Git itself. Notably, Git itself comes with two visual tools, **gitk** [4] and **git-gui** [5]. The former is used mainly for viewing the commit history, and the latter's main function is to create commits [6]. However, these tools provide minimal 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 the 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 many complex ways through the desktop application *SourceTree* [7]. 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 2.1 is an example of this.

    The project, therefore, aims to build upon the positive features of SourceTree and the Atlassian ecosystem, adding more abstraction and automation in the hopes of achieving a much simpler user experience.
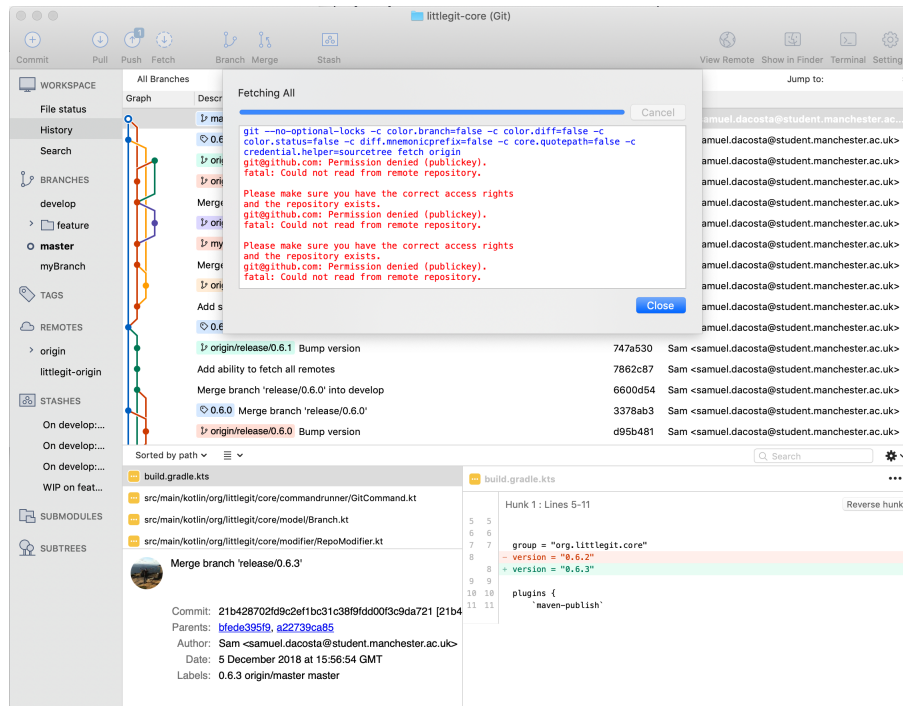
Figure 2.1: An example of SourceTree passing errors on to the user.

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 2.2 shows a high-level representation of the system. The desktop GUI is the application the end user installs and uses to manipulate their repositories.

The Git servers are the machines that host these remote repositories. These are distinct from the main server which does *not* host repositories but manages the system as a whole.

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, using these limits our ability to automate 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 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 2.2 shows multiple Git servers; this demands some explanation. A reasonable question to ask is why is one not sufficient? The second objective of this project is to produce a scalable system. We must ask ourselves what
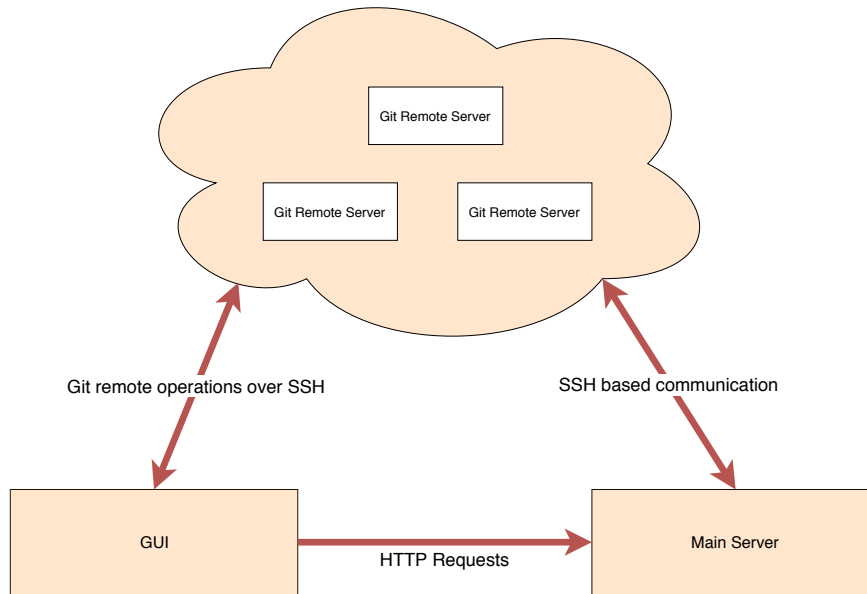
Figure 2.2: A high level representation of the architecture of the system.

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. 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 main server provides API endpoints for the GUI to use to handle all user authentication, create new repositories and importantly, to register new SSH keys.

2.2.2 Desktop 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. It also is responsible for the aforementioned automation of Git and resolving of issues.
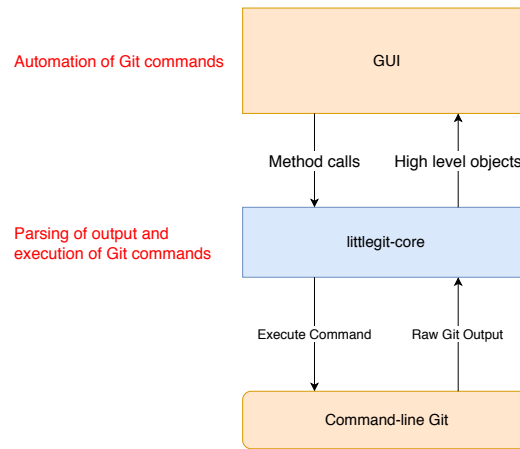
Figure 2.3: A representation of the interaction between the GUI and the littlegit-core

Low-level git library

At this stage, it's prudent to mention the way both the GUI and the main server interact with Git itself. This interaction happens through the medium of a library called the **littlegit-core** written for this system and included by the GUI and server as a dependency. 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 as Figure 2.3 describes.

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 applications SourceTree, GitKraken [8] and GitK. Figures 2.4, 2.5 and 2.6 show a screenshots of their main interfaces. The first thing to note in looking at the interfaces is the common element being the Git graph. Though each has a different style and displays the information differently, the graphs are all prominent 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 necessary information, but is easy to present a simplified version (as was needed for littlegit) by merely omitting some of the information surrounding it.

However, other options were considered, Figure 2.7 shows an early design for an alternative graph which scrolls horizontally. 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
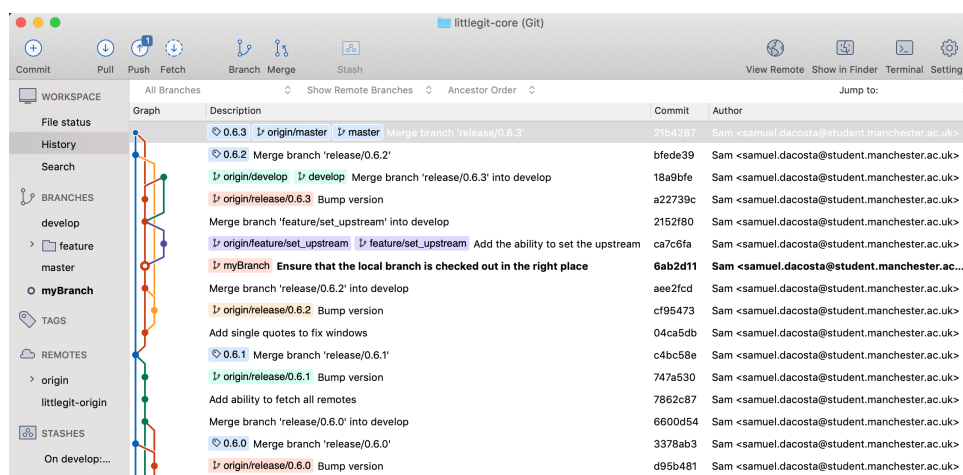
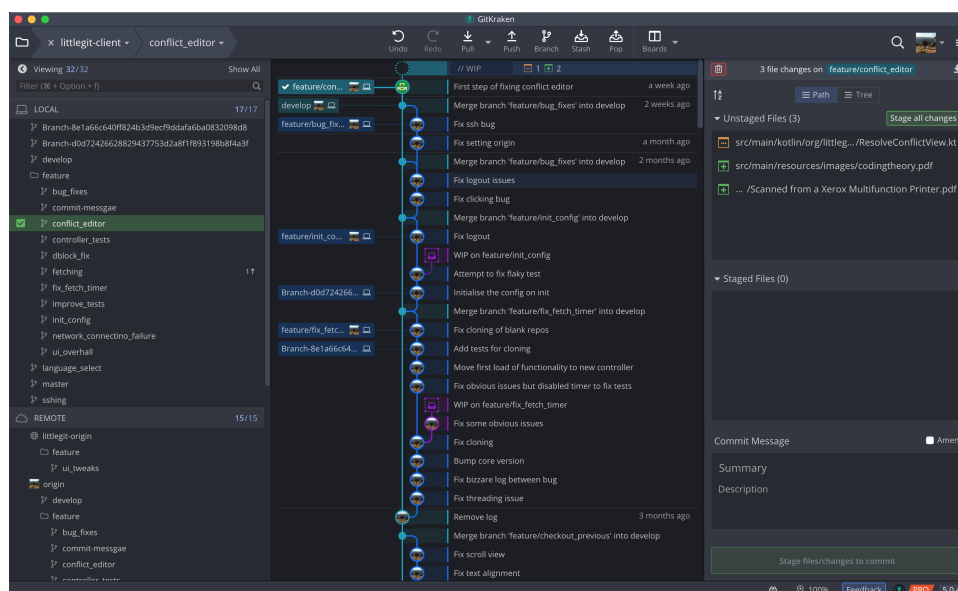Figure 2.4: A screenshot of the main interface of the application SourceTree.

Figure 2.5: A screenshot of the main interface of the application GitKraken.
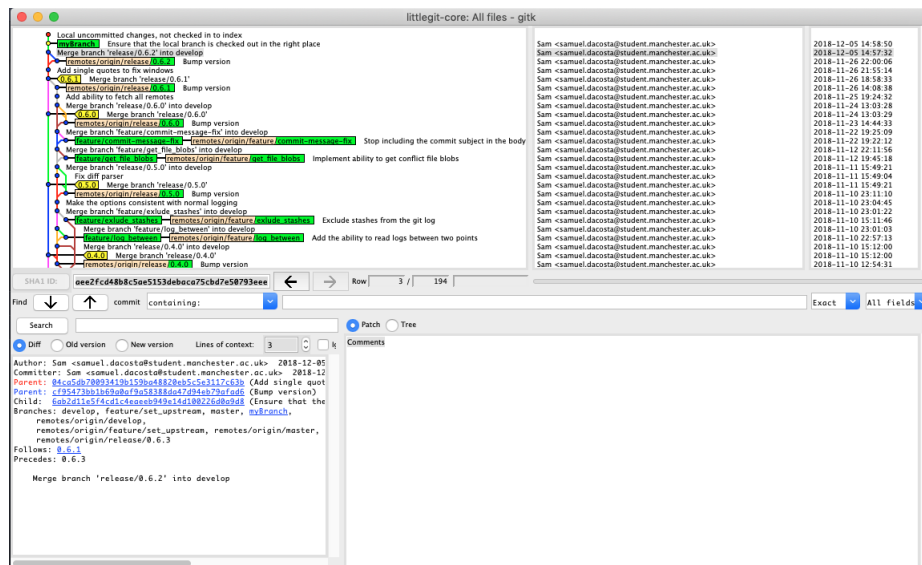
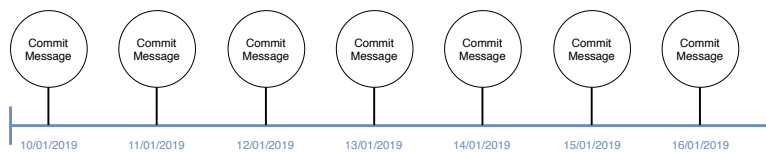Figure 2.6: A screenshot of the main interface of the application GitK.

Figure 2.7: An early design for the Git graph using horizontal scrolling.

idea however soon proved impractical, visually representing branches in this model becomes messy and complicated, especially for more complex repositories. Furthermore, according to user testing [9], 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.

Moreover, all three applications have lots of buttons and menus. After studying these controls, 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 to use a popup dialogue to indicate 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 than in other workflows, only being used by littlegit internally when needed. Looking back at SourceTree especially, it is unclear how to create a new commit in its interface. Since commits drive littlegit, it was imperative to avoid this.

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 switch to 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. Use of the graph for navigation seemed an efficient and intuitive approach. With littlegit we aimed to expand this functionality, allowing the user to check out any commit in the history.

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. As a result, there is 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 research conclusions are useful, particularly a study by the Neilsen Norman Group concerning designing for teenagers [10]. 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 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 the importance of fast loading speeds. Though this refers specifically to websites, the study highlights the impatience of teenagers while using software. Therefore, the second objective of the project 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 [11] was used to produce initial designs. These designs are included in Figures 2.8 and 2.9.
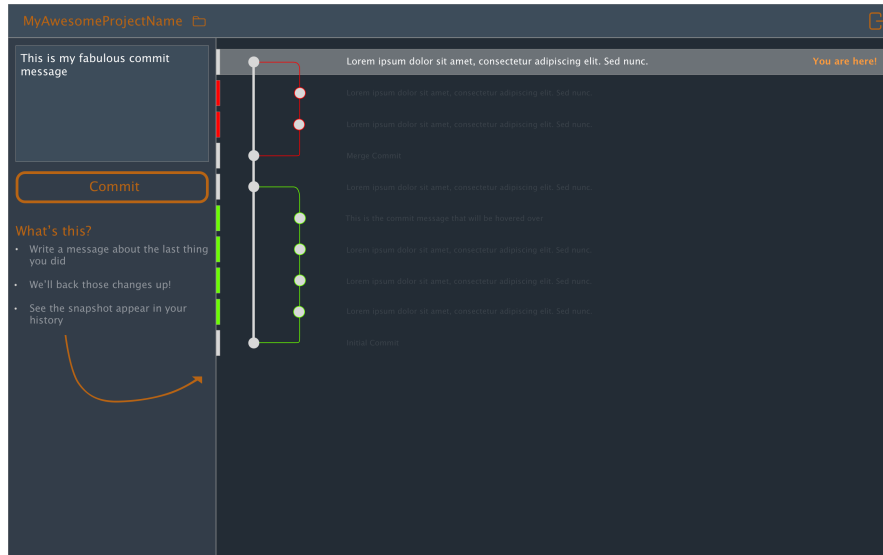
Figure 2.8: A design for the main screen of littlegit
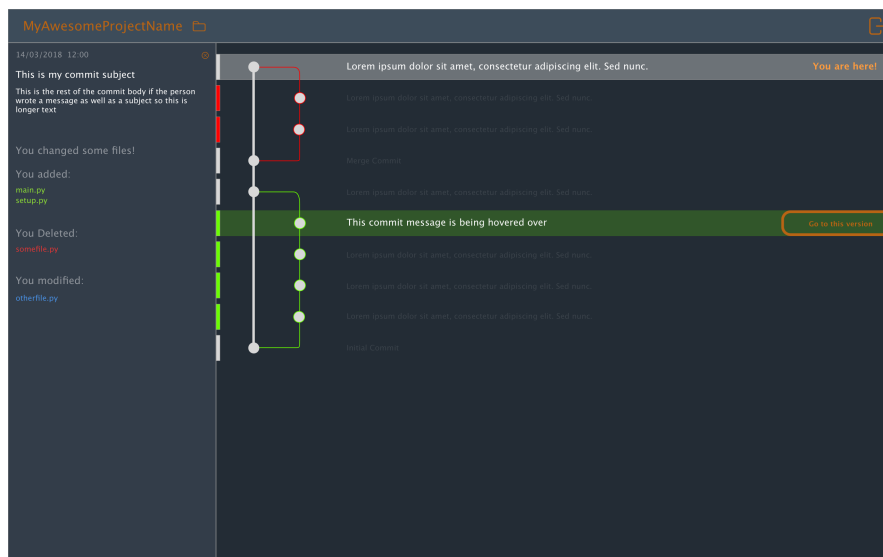
Figure 2.9: A design for the main screen of littlegit after a user clicks a commit from the graph.

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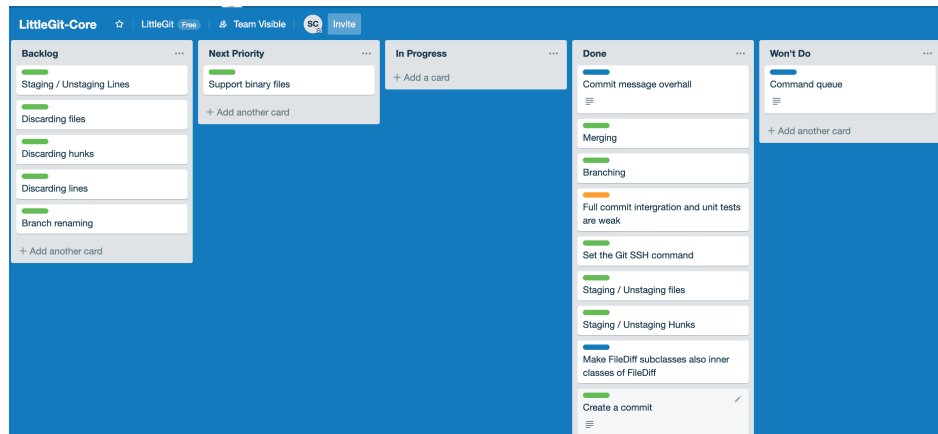
Figure 3.1: A screenshot of the Trello board for the littlegit-core component of the project, taken after the end of development.

3.1 Planning and management of work

Trello boards were used from the very beginning to keep track of project goals and tasks. Separate boards kept track of tasks relating to the server, the littlegit-core and the GUI since they are three distinct codebases. 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.

3.2 Choice of language

Considering the time constraints on this project, only implementing the littlegit-core once, was crucial. Furthermore, the time constraints would not permit writing further wrappers around 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, meaning they needed to use the same language. 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 [12] has become a popular technology allowing JavaScript, HTML and CSS (traditionally web technologies) to be used to produce cross-platform desktop applications.

   Furthermore, NodeJS [13] 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 personal 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 [14] 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 which is fully compatible with Java frameworks and libraries enabling longstanding, reliable frameworks to be used.

   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 work with. 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.

Moreover, the Tornado Fx [15] framework is a relatively new, open source framework which allows the development of cross-platform (Linux, Mac and Windows) desktop applications in Kotlin.

3.3   The littlegit-core

As has already been discussed, this is the library that interacts directly with the Git binary. The primary challenge here was ensuring we use Git-plumbing commands since these are designed for machine use. A further discussion of Git-plumbing is included in Appendix B.

The project aims to follow modern best practices for software development. For this reason, a TDD (test driven development) approach was used throughout. To achieve this, the library was *designed for testibility*. To aid testability, the library's operations were made synchronous. Methods which return synchronously prove to be much simpler and more reliable to test, allowing the entire library to be covered thoroughly with tests.

3.3.1   Architecture

The GUI uses the littlegit-core to execute Git commands on local Git repositories. However, the main server needs to be able to execute commands on remote servers over an SSH connection. This requirement is a primary driver behind the architecture of the littlegit-core which visualised in Figure 3.2.

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 into 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 for the server.
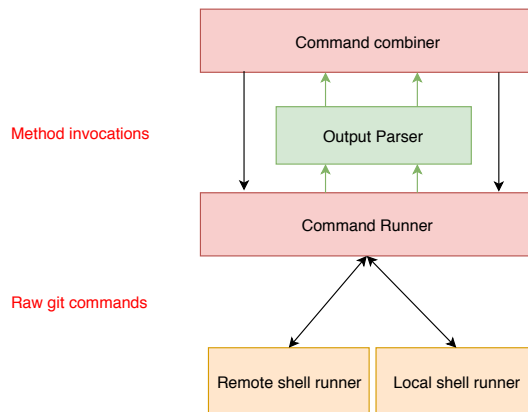

Figure 3.2: The internal architecture of the littlegit-core.

3.3.2 Deployment

We keep mentioning that the library is *used* or *included* by the GUI and server. Figure 3.3 shows the deployment pipeline which concludes by deploying the library to a Maven [16] 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. The deployment is done using the Travis [17] continuous integration service (which is also used to deploy and run tests for the other project components).
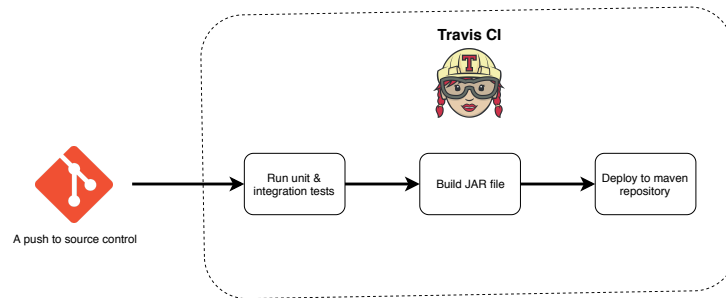
Figure 3.3: The deployment pipeline of the littlegit-core.

3.4 The main server

In this section, we will explore the implementation of the main server and justify the technologies used.

3.4.1 Technologies

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 for students, AWS offers a free tier of the services needed. We will now explore the different technologies considered for components that make up the server.

Web server framework

A web server framework was needed as a starting point to build the API. The two frameworks most seriously considered are as follows.

- The Spring framework [18].

- The Jersey framework [19].

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 implemented 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 [20] and Firebase [21], 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.

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 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.

Redis [22] was selected as the caching solution, its feature set is vast, 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 letting the cache speed up common simple operations such as returning the details for a particular user, or checking if an authentication token is valid. Moreover, having a cache with built-in replication allows it to scale across multiple machines easily giving
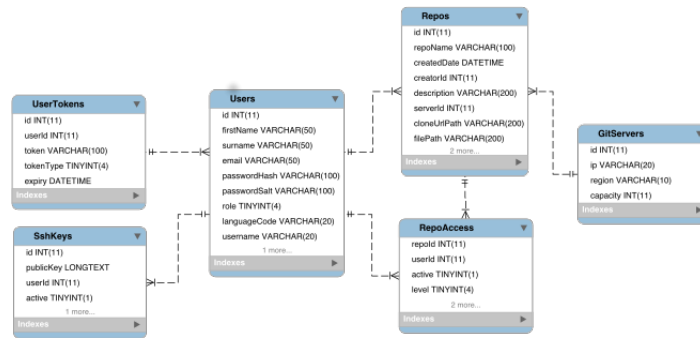
Figure 3.4: An entity relationship diagram representing the database.

us the horizontal scaling we require. The reader may wonder why we are using a separate caching solution rather than making use of data structures within the server code itself to achieve the same thing.

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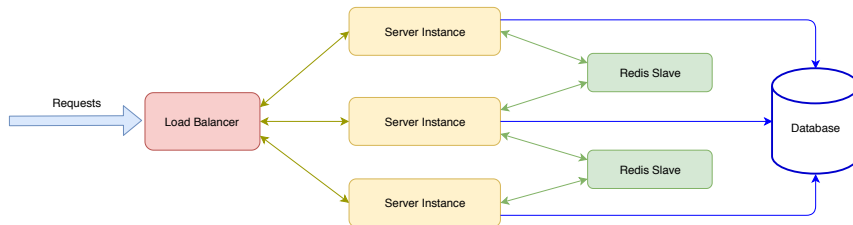
Figure 3.5: The overall architecture of the main server and the way it is designed to scale.

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 a machine while also being straightforward to understand by humans (making debugging easy) [23].

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.

---

[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, the code however fully supports the architecture described.

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 architecture chosen was a *Layered Architecture* as shown in Figure 3.6. Each layer abstracts the one beneath it and interacts exclusively with that layer; this seperation makes testing very straightforward. The roles of the individual layers are described below.
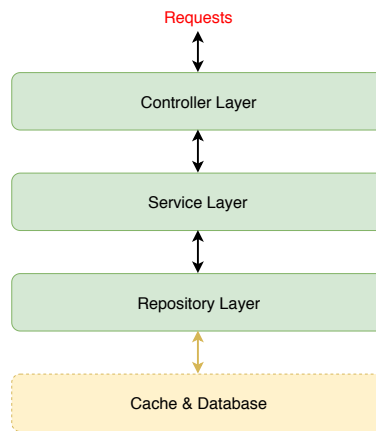
Figure 3.6: The internal architecture of the main server.

Repository Layer

The repository layer (not to be confused with a Git repository) interacts directly with the database and cache. This involves using raw SQL commands to communicate with the database and is responsible for populating the cache and invalidating cached information when data is updated. 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.

Furthermore, the repository layer handles the mapping of data from the database and cache to Kotlin data structures. In short, the repository layer is an abstraction on the database and cache, allowing the service layer to work at a level above the raw SQL.

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 challenging 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 so that the bulk of unit testing can be done against the service layer.

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 1.2, 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 (a fast calculation). We then only draw the visible portion, rather than drawing everything and letting the window clip out what is out of frame.

Experimentation with the resulting algorithm has shown that the time to redraw the graph is now negligible in comparison with the time taken to read the history from Git in order to draw it (an unavoidable step).
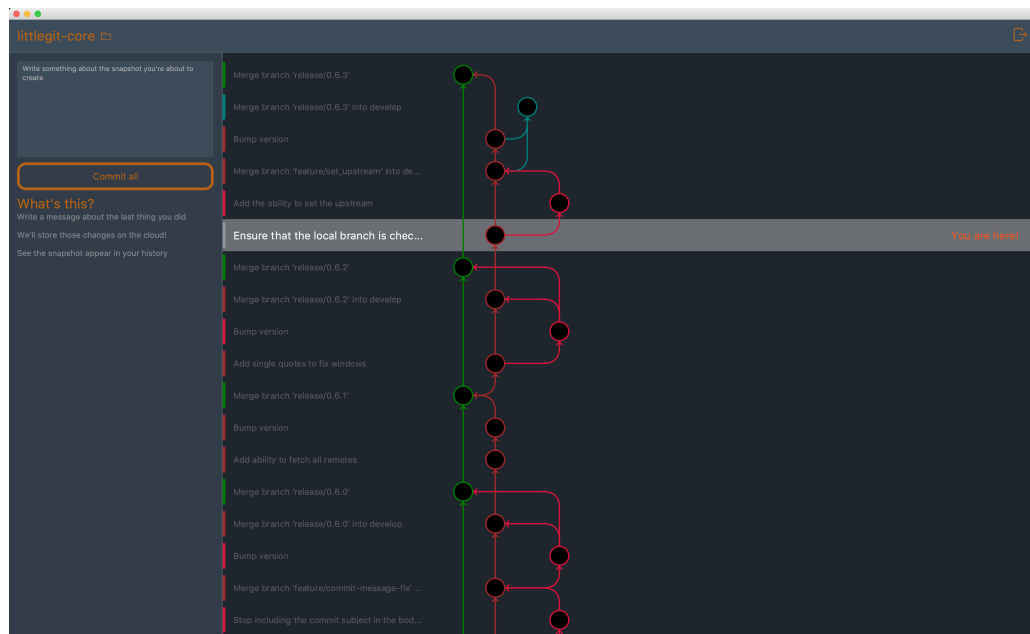
Figure 3.7: The Git graph for the littlegit-core library opened in littlegit.

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.

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. 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 [24]. Furthermore, 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 runs, 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** [25] which enforces 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 necessary 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.

Shell commands are used on the Git servers, invoked by SSH to check with the main server if a specific user action is allowed before completing it. This ensures the main server controls repository access allowing access to be easily revoked from users if needed. Appendix C describes details of the full authentication process.

Chapter 4

Evaluation and Testing

This Chapter discusses the approach taken to test and evaluate the system. We'll discuss 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 primary aim of user testing was not on finding bugs due to the limited amount of time available interacting with likely users (i.e. teenagers). It was decided that obtaining feedback about the usability and usefulness of the system was a better use of this time, rather than having users try to break the system to discover faults.

4.1   Automated Testing

This project consists of three distinct codebases. Both the littlegit-core and desktop GUI are required to work cross-platform, as such, all the automated tests were run in OSX, Windows and Ubuntu-Linux environments. A test-driven approach was used across the board but 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** [26]. It stipulates that a large proportion of the tests should be unit tests with a smaller proportion of integration 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 implement these tests as unit tests, we would have to mock Git's output at each stage of the test. Such an approach quickly proves 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 use integration tests 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.

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

Coverage: Run all ×

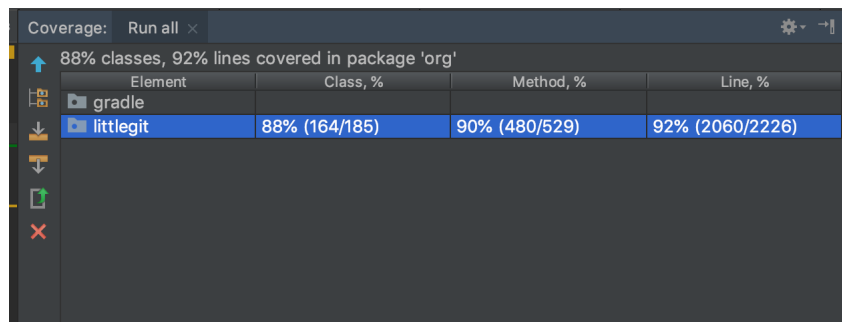88% classes, 92% lines covered in package 'org'

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

In practice, however, the suite of over a hundred tests still runs (on average) in under 15 seconds 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 discovered.

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 (similarly for the *Controller* layer). A much smaller number of integration tests are used to check the entire stack from the *Controller* layer down to the database, the aim being to catch any issues in the integration of the various layers. Code coverage for the server is lower than the littlegit-core but still stands at 76% which is acceptable.

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 was deemed unnecessary.

Improvements

Manual testing was used to sanity check the API endpoints. Involving running the server, and then calling the endpoints with input data and reviewing the JSON response. Automating this process would be useful, running tests 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 to be tested run asynchronously. The strategy, therefore, was to
concentrate on the code that performs automation of Git operations, 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.

It was decided that UI tests would not be used since the Tornado FX frame-
work's support for them is limited, and early experimentation showed them to
be unreliable. Due to the lack of 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 with teenagers occurred during development,
the first roughly halfway through and the second nearing the end. The partic-
ipants in the sessions were all sixteen years olds studying GCSEs in Computer
Science. They had not used any version-control systems in the past but had
received a short talk about Git specifically before the feedback session.

First feedback session

During the first session, the software was unfinished with a few bugs in the way
repositories were created and opened. To mitigate this, a new (empty) reposi-
tory was opened in the software beforehand. Questions were asked to the group
as a whole, while tasks were carried out by a single individual. The questions
and tasks along with a summary of responses to them are as follows.

Question: What are your general first impressions of the application?
Summary of responses: General opinion was favourable; participants liked the
dark colour scheme and the uncluttered interface. One participant commented
that the limited amount of text made it much simpler to use by those with
dyslexia like themselves.

Task: Could you make a change to a sample file and commit those changes?
Observations: It was very encouraging to see the task was completed smoothly
without any issues. Even more encouragingly the user then clicked on the com-
mit to examine its details without being prompted to do so. When asked, they

commented that it seemed intuitive to use the graph to view details of the commit.

Task: Could you create another couple of commits and then go back to the first version?
Observations: Again, this task was carried out smoothly. Performed by a different person to the first task, they added a few commits and took very little time exploring the interface before finding the option to switch back to an old version. Another member of the group who was observing commented that they liked the fact they could use the graph for navigating back and forth.

Question: For this question, participants were presented with the interface to GitKraken (discussed in Chapter 2), they were asked for any comments they had comparing the two applications.
Summary of responses: The overwhelming reaction to GitKraken's interface was that it was difficult to navigate and to understand what was going on. Many of the options and buttons (including branching and merging) were explained to participants and most commented that they didn't see such features as being very useful in their own workflows.

   Overall, the consensus was that for their current needs and level of experience that the simplified interface of littlegit was preferable to the more sophisticated interface of what we might consider a more conventional Git GUI. This piece of feedback was particularly useful since it indicates that we did not *over-simplify* Git to the extent it stopped being useful.

Overall comments: One of the main takeaways from this first session was that while obtaining the positive feedback was indeed reassuring, for the next session it was necessary to try to push for more criticism. In other words to make participants comfortable in expressing suggestions and pointing out issues they saw in the software.

Second feedback session

The format of the first session worked well and was therefore used for the second session. The second group made up of some new faces and some of the same participants as the first group; however, the main difference was that before starting, criticism and negative comments were encouraged. The questions, tasks and observations are as follows.

Task: Could you open the sample directory in littlegit?
Observations: The user (intentionally chosen to be someone who hadn't been in the previous group and hadn't seen the software before) successfully opened the indicated directory in the software. They were then asked to create a few commits and navigate back and forth through the commit history as in the previous session and had no issues doing so.

Task: Could you open one of the existing repositories that only exist on the server?
Observations: This task caused some confusion, and it was necessary to explain further what was meant by repositories not yet existing locally. The user was
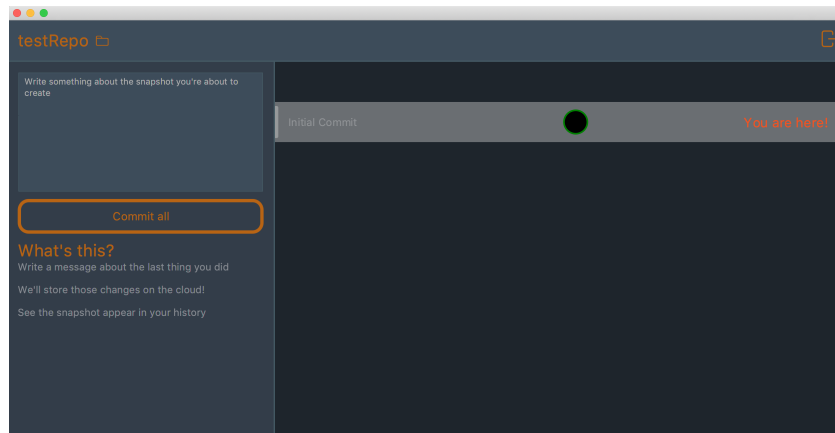
Figure 4.2: The main interface of littlegit with only one commit.

able to open the repository in littlegit but was unable to locate it on the file system; a clear need for improvement in the software. Participants suggested a button to open the directory in the file explorer which would be an excellent solution to the problem. Further work is certainly needed in this area to improve the user experience.

Overall comments: During the session, one of the participants mentioned that the Git graph in the software looked odd when it only had one commit. Others agreed, and it is easy to see why. The commit is highlighted but has no other commit to distinguish it from which looks odd. Figure 4.2 shows the software in this state.

Unfortunately, time constraints prevented these concerns from being adequately addressed during the main project lifecycle, but these are indeed areas for future improvement, making the feedback valuable.

Other user feedback

Other, less formal feedback sessions were carried out numerous times within a small tutor group involving university students and lecturers where feedback was obtained about the project's progress. Everyone in the room had experience using Git, and therefore the format was slightly different. No tasks were set; instead, the features of the software were briefly demonstrated and feedback gathered. 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 rationale 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. The most significant change made if this project were to be restarted is 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 Java FX online is minimal, making development more difficult and more importantly making the software tough to distribute and install. Installing the software on machines in schools is highly unlikely to be feasible, which is a major disadvantage to software aimed at teenagers. In hindsight, despite the massive disadvantages of JavaScript as a language, use of the Electron framework which we disregarded in Chapter 3, may have been the better choice to address these issues.

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 and 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 multiple threads) and am very happy with the results! On a related note, security and authentication (especially using OAuth 2.0) 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 decisions to use them follow directly from the decisions discussed in Chapter 3, or from merely being industry standard libraries for their given functions.

Retrofit [27] was used for performing API calls from the desktop GUI to the main server, and Moshi [28] used for the conversions between JSON to Kotlin objects. For communication between the main server and the Redis cache, Jedis [29] was used, chosen for its speed and simplicity.

Dependency injection was used in architecting the main server code, improving readability, maintainability and testability. This technique makes the architecture described in Chapter 3 feasible. The dependency injection framework chosen was Dagger 2 [30].

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 [31] under the Creative Commons licence.

Appendix B

Git commands

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.

Plumbing commands, on the other hand, are generally lower level commands to the porcelain and are often used internally 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 the invocation of multiple plumbing commands. The littlegit-core endeavours to use plumbing commands wherever possible to ensure compatibility with future Git versions.

Appendix C

Git authentication

To explain the way littlegit handles Git authentication 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. 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 each of these servers' `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. The main server chooses one of the Git servers from the available pool and sets up the remote upon it by executing commands via SSH (making use of the littlegit-core). This mechanism also makes it easy to add new Git servers to the available pool, which will immediately be picked up and used by the main server.

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. 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 public 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 if not.

Unfortunately, this presents us with a potential performance bottleneck. The list is searched sequentially by SSH 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 holding only a subset of the total SSH keys leads to far fewer lines in each `authorized_keys` file.

Bibliography

[1] Ofqual. *GCSE outcomes in England.* `https://analytics.ofqual.gov.uk/apps/2018/GCSE/Outcomes/`.
Accessed: 13-03-2019.

[2] Department of education. *Computing programmes of study.* `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/239033/PRIMARY_national_curriculum_-_Computing.pdf`.
Accessed: 02-03-2019.

[3] Linus Torvalds. *Git.* `https://git-scm.com`.
Accessed: 26-02-2019.

[4] Git. *gitk.* `https://git-scm.com/docs/gitk`.
Accessed: 26-02-2019.

[5] Git. *git-gui.* `https://git-scm.com/docs/git-gui`.
Accessed: 26-02-2019.

[6] git. *Git in Other Environments - Graphical Interfaces.* `https://git-scm.com/book/en/v2/Appendix-A%3A-Git-in-Other-Environments-Graphical-Interfaces`.
Accessed: 26-02-2019.

[7] Atlassian. *SourceTree.* `https://www.sourcetreeapp.com/`.
Accessed: 26-02-2019.

[8] axosoft. *GitKraken.* `https://www.gitkraken.com/`.
Accessed: 28-02-2019.

[9] Jakob Nielsen. *Scrolling and Scrollbars.* `https://www.nngroup.com/articles/scrolling-and-scrollbars/`.
Accessed: 26-02-2019.

[10] Alita Joyce and Jakob Nielsen. *Teenagers UX: Designing for Teens.* `https://www.nngroup.com/articles/usability-of-websites-for-teenagers/`.
Accessed: 26-02-2019.

[11] Bohemian Coding. *Sketch.* `https://www.sketch.com/`.
Accessed: 26-02-2019.

[12] *Electron.* `https://electronjs.org/`.
Accessed: 26-02-2019.

[13] *NodeJS.* `https://nodejs.org/en/`.
     Accessed: 26-02-2019.

[14] Microsoft Corporation. *ASP.NET.* `https://dotnet.microsoft.com/apps/aspnet`.
     Accessed: 14-04-2019.

[15] Edvin Syse. *Tornado FX.* `https://tornadofx.io`.
     Accessed: 14-04-2019.

[16] Apache. *Apache Maven.* `https://maven.apache.org/`.
     Accessed: 02-03-2019.

[17] *Travis CI.* `https://travis-ci.com/`.
     Accessed: 28-04-2019.

[18] Pivotal. *The Spring framework.* `https://spring.io/`.
     Accessed: 14-04-2019.

[19] Oracle Corporation. *Jersey.* `https://jersey.github.io/`.
     Accessed: 14-04-2019.

[20] Apache. *CouchDB.* `http://couchdb.apache.org/`.
     Accessed: 02-03-2019.

[21] Google. *Firebase.* `https://firebase.google.com/`.
     Accessed: 02-03-2019.

[22] redislabs. *Redis.* `https://redis.io/`.
     Accessed: 14-04-2019.

[23] W3Schools. *What is JSON?* `https://www.w3schools.com/whatis/whatis_json.asp`.
     Accessed: 14-04-2019.

[24] Aaron Parecki. *OAuth 2.0.* `https://oauth.net/2/`.
     Accessed: 14-04-2019.

[25] Git. *Git shell.* `https://git-scm.com/docs/git-shell`.
     Accessed: 02-03-2019.

[26] Ham Vocke. *The Practical Test Pyramid.* `https://martinfowler.com/articles/practical-test-pyramid.html`.
     Accessed: 02-04-2019.

[27] Square. *Retrofit.* `https://square.github.io/retrofit/`.
     Accessed: 14-04-2019.

[28] Square. *Moshi.* `https://github.com/square/moshi`.
     Accessed: 02-03-2019.

[29] Jonathan Leibiusky. *Jedis.* `https://github.com/xetorthio/jedis`.
     Accessed: 02-03-2019.

[30] Google. *Dagger 2.* `https://github.com/google/dagger`.
     Accessed: 02-03-2019.

[31] Icongeek26.   *Flat   Icon.*   `https://www.flaticon.com/authors/`
     `icongeek26`.
     Accessed: 02-03-2019.