

StarFLOSS

An observatory of FLOSS Communities

Camila Naomi Kodaira, Felipe Caetano Silva

CAPSTONE PROJECT PRESENTED TO THE
INSTITUTE OF MATHEMATICS AND STATISTICS
OF THE UNIVERSITY OF SÃO PAULO
IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR IN COMPUTER SCIENCE

Program: Computer Science

Advisor: Paulo Meirelles

Coadvisor: Melissa Wen

São Paulo

November 20th, 2019

StarFLOSS

An observatory of FLOSS Communities

Camila Naomi Kodaira, Felipe Caetano Silva

This is the original version of the capstone
project prepared by the candidates Camila
Naomi Kodaira, Felipe Caetano Silva.

*Given enough bots, all behaviors
are documented and visualized.*

Acknowledgements

Camila Naomi Kodaira

This project would not have become what it is without the help of many different people, and for that I would like to offer my thanks to all of them.

To the students of the *MAC0413 class of 2019*. For helping to materialise a proof of concept for this project, from which ideas were created and improvements were made.

To *Ricardo Marques*, the Valet at iTower. For hearing me rant about issues I was having with this project and encouraging me to finish it, for almost four months, even if he had no coding background.

To my parents, *Elisa Tiemi Hasegawa Kodaira* and *Clóvis Yoshio Kodaira*. For believing in me and providing good advice for balancing my professional, educational and personal life in a way that no one thing swallowed the other.

To my colleagues in HP. For encouraging and allowing me to prioritise writing the capstone project above all else, even if that meant doing it on working hours, and for talking about their past experiences, showing me all my concerns are common and even with them everything will be okay.

Special thanks to both our advisers, *Paulo Meirelles* and *Melissa Wen*. Without their guidance this project would not have been at all, they gave us hope, encouragement and a positive view of the future on the most stressful moments.

And to *Felipe Caetano Silva*. I can not think of going on this journey alone, the problems we had we dealt with together, having someone else to depend on when things got difficult made me continue this journey, no matter how long and winding the road.

*Even if its petals scatter,
a peony is still a flower
Even if the summer ends, the
memories of it are still cherished
— Yorushika, Itte*

Felipe Caetano Silva

Under no circumstances would this project culminate in this moment if I had not received help when in need, for every little help I humbly offer my thanks to all.

To *Lucas Henrique Nascimento*. For all the support and being the first person to listen to my crazy idea of coming to this University.

To my first inspiring Fundamental Teacher *Derli Francelli*, my first chess teacher. I would have never learned how to focus if it was not for her teaching and caring.

To my Parents *Reginaldo Caetano Silva* and *Josefa Gomes da Silva* and my brother *Jefferson Caetano Silva*. For all motivational speeches and all the sacrifices made in order to help me graduate.

To all my friends from the *Ducks*. For every bit of information given by them, for every piece of talk to forget all the problems.

To all the Professors I have encountered here. They presented a new world to my narrow little mind, making it expand exponentially.

A special thanks to *Paulo Meirelles and Melissa* our advisers. Their ideas always clarified the darkest moments, made us believe in ourselves and guided us through this long path.

To *Camila Naomi Kodaira*. Thanks for all we passed together. Everything was amusing even the troubles we faced. I would have reached here if it was not for your realist positivism.

*Water can flow or creep or drip
or crash. Be water, my friend.
— Bruce Lee*

Resumo

Camila Naomi Kodaira, Felipe Caetano Silva. **StarFLOSS**. Dissertação (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

Free/Libre/Open Source Software(FLOSS), em português chamado de software livre, é um termo guarda-chuva para softwares em que seus usuários são livres para usar, mudar, estudar e redistribuir seu código de forma livre. Por mais de duas décadas, muitos estudos procuram uma forma atraente de descrever as práticas adotadas em projetos de software livre bem sucedidos. Comum a esses projetos está a infra-estrutura utilizada para comunicação dos contribuidores, geralmente são Wikis, Listas de email, Canais de Internet Relay Chat(IRC), Repositórios, Websites e bug trackers. Nesse contexto, o projeto StarFLOSS tem como objetivo observar os canais existentes para coletar informação e apresenta-las de forma coesa, divertida e fácil. O projeto consiste de três camadas principais que permitem as duas funcionalidades. A primeira camada é o Website, por ser a camada de entrada dos usuários ela foi cuidadosamente planejada enfatizando uma identidade visual prazerosa de se olhar e fácil de usar. Cada parte de sua Dashboard é interativa, isto é, a seleção de qualquer parte dos gráficos e menus acarreta uma filtragem dos dados apresentados. Essa camada foi criada utilizando React como o framework para o Website e a biblioteca D3 para o desenho dos gráficos. Todas as componentes foram feitas utilizando HTML e CSS puro. A próxima camada é a de Integração, que é representada por um Gateway e um Agregador de Dados. Entre suas responsabilidades estão a de fazer com que todas as requisições dos clientes para outros serviços passem por seu Gateway, dando assim a impressão de que a plataforma inteira funciona em um único módulo. A terceira camada são os coletores. Eles representam os microserviços utilizados para reunir os dados das fontes e fornecer aos clientes. São ferramentas auto-suficientes e que podem ser usadas para qualquer outro projeto. A arquitetura da plataforma foi baseada em SOA (*Service Oriented Architecture*). Esse estilo de design de software e os seus princípios centrais são independentes de produtos e tecnologias. Ao final deste projeto, conseguimos construir a fundação de um projeto que possibilita usuários não programadores observar comunidades de software livre. Utilizando práticas que facilitam a integração de outros contribuidores possibilitamos que o projeto tenha continuidade e também seja um software livre por si só.

Palavras-chave: StarFLOSS. API Gateway. Visualização de dados. Software Livre.

Abstract

Camila Naomi Kodaira, Felipe Caetano Silva. **StarFLOSS: *An observatory of FLOSS Communities***. Capstone Project (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

Free/Libre/Open Source Software (FLOSS) is an umbrella term for software in which its users are free to use, change, study, and redistribute freely. For over two decades, many investigations have looked for an attractive way to describe the practices adopted in successful open source projects. Common to these projects is the infrastructure used for contributor communication, usually Wikis, Mailing Lists, Internet Relay Chat (IRC) Channels, Repositories, Websites, and Bug Trackers. In this context, the StarFLOSS project aims to observe these channels while collecting information and present it in a cohesive, fun, and easy way. The project consists of three main layers that allow these functionalities. The first layer is the Website. It is the entrance to new users; it has been carefully designed to emphasise a visual identity that is pleasant to look at and easy to use. Each part of its Dashboard is interactive, that is, selecting any part of the graphs and menus entails filtering the data presented to illustrate the selection. This layer was created using React as the website framework and the D3 library for graphics design. All components were made using pure HTML and CSS. A Gateway and a Data Aggregator represent the Integration Layer. Their responsibilities include making all customer requests for other services pass through their gateway. Thus, giving the impression that the entire platform works in a single module, it was built mainly using the Ruby on Rails framework. The third layer is the collectors. They represent the microservices used to gather data from sources and provide to clients. They are self-contained tools that can be used for any other project. The platform architecture was based on Service Oriented Architecture (SOA). This style of software design and its core principles are independent of products and technologies. At the end of this project, we were able to build the foundation of a project that enables non-programmer users to observe open source communities. Moreover, we developed it using practices that facilitate the integration of other contributors and also allowed the project to continue and be free software by itself.

Keywords: StarFLOSS. API Gateway. Data Visualisation. Free Software.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivations and objectives | 2 |
| 1.2 | Development Methodology | 2 |
| 1.3 | Organisation of the capstone project | 4 |
| 2 | Background | 5 |
| 2.1 | Free/Libre/Open Source Software | 5 |
| 2.2 | Git | 7 |
| 2.3 | Related Projects | 8 |
| 2.4 | StarFLOSS features | 9 |
| 2.4.1 | Front-end features | 10 |
| 2.4.2 | Back-end features | 10 |
| 3 | The development of an Observatory of FLOSS Communities | 13 |
| 3.1 | The user Interface | 13 |
| 3.1.1 | Creating a visual identity | 13 |
| 3.1.2 | The importance of visual design fundamentals | 15 |
| 3.1.3 | A solid website with React | 18 |
| 3.1.4 | An interactive dashboard with D3.js | 19 |
| 3.2 | The Integration Layer | 23 |
| 3.2.1 | Architecture | 24 |
| 3.2.2 | Good Practices | 26 |
| 3.2.3 | Restful API | 27 |
| 3.2.4 | Ruby on Rails | 27 |
| 3.2.5 | Documentation | 28 |
| 3.2.6 | The Commit Bot | 29 |
| 3.2.7 | The IRC Bot | 31 |
| 3.2.8 | Email Lists | 32 |

| | | |
|----------|---|-----------|
| 4 | The Git Community in StarFLOSS | 33 |
| 4.1 | The home page | 33 |
| 4.2 | The dashboard | 34 |
| 4.2.1 | Information menu | 35 |
| 4.2.2 | Members menu | 35 |
| 4.2.3 | Time of Day - Heatmap | 36 |
| 4.2.4 | Commits over time - Bubble chart | 37 |
| 5 | Final Remarks | 39 |
| 5.1 | Where to go from here? | 39 |
| 5.1.1 | The future of the StarFLOSS website | 40 |
| 5.1.2 | The future of the StarFLOSS API | 40 |

Appendices

| | | |
|----------|--|-----------|
| A | Flux pattern | 43 |
| B | Endpoints | 45 |
| B.1 | Commit Information Endpoints | 45 |
| B.2 | Email Information Endpoints | 46 |
| B.3 | IRC Information Endpoints | 48 |

Annexes

| | |
|-------------------|-----------|
| References | 51 |
|-------------------|-----------|

Chapter 1

Introduction

Free/Libre/Open Source software (FLOSS) is an umbrella term for software in which users are free to inspect, use, change, and distribute it (CROWSTON, WEI, et al., 2012). The singularities of FLOSS project management and the dynamics of their communities are topics extensively discussed in Software Engineering studies over the last decade, as in SCACCHI, 2010; KON et al., 2011; CROWSTON, WEI, et al., 2012; STEINMACHER et al., 2015; CROWSTON and SQUIRE, 2017; BARCOMB et al., 2019.

For over two decades, investigations have sought to describe better the practices surrounding a successful FLOSS project. In 1999, Raymond published “The Cathedral and the Bazaar” (RAYMOND, 1997), defining the development model of free software as a Bazaar. In this seminal essay, the author coined the following statement: “Given enough eyeballs, all bugs are shallow”, which became known as the “Linus Law”. This statement covers one of the aspects of the development of FLOSS: the voluntary nature of its contributors. In this way, contributors are expert system users, and once they find bugs, they make efforts to report and even repair the bugs by themselves.

For long, Raymond’s essay has influenced the software engineering community as an ideal way of producing software. However, over the years, technology has enhanced, and the way to develop software has evolved. As a result, interactions in FLOSS communities have also changed. More than changes, FLOSS projects today feature a diversity of approaches to development management and community organisation.

In hindsight, the idea of homogeneity of development, where all FLOSS use the same development methodology and techniques as the ones used in the prestigious projects, is inaccurate (ØSTERLIE and JACCHERI, 2007). Another misconception is thinking that every community would behave like the ones usually used as examples, like the Linux Kernel and the GCC projects, both successful with many contributors and different channels of communication (ØSTERLIE and JACCHERI, 2007). Lastly, there is also a misunderstanding surrounding the voluntary aspect of the contributions. Currently, many companies considered big players in the commercial world also participate in the development of FLOSS code, and the question of whether their specific contributions would be convenient to the project still lingers (FITZGERALD, 2006).

Given this context, it becomes essential to observe the dynamics of the current FLOSS

projects, so that it is possible to grasp what is happening in the diverse set of their developments.

1.1 Motivations and objectives

With FLOSS being a complex world with many different projects, it can be hard for anyone outside and inside their given community to see what is going on in the grand picture. That means it is complicated to measure how efficient this development process really is, and how it can change through time.

The main reason why it is not easy to track down the different communities is the many different communication methods that they may be using. In Section 2.1 we describe all these methods with more detail. For example, the Linux kernel has a repository on GitHub where issues and commits are tracked. However, to be able to make a commit in Git, the programmer first sends an email on a specific email group, with the code to review. Also, any minor conversations are handled in a separated IRC chat. Those communications setups become more convoluted with the growth of the community, and there is no set pattern to any of them. Consequently, if tracking a single FLOSS project is troublesome, monitoring the overall state of FLOSS is not a simple task.

An alternative to overcome the obstacle of not having unified channels for communication in FLOSS communities is to rely on the support of those who are already familiar with a given project. If those who know what the communication method of their community is, create a list of what is most important to track, the work that is left is relatively simple. Thus, it is possible to create a tracker for those methods and a place that can unify that data and display it. With so many projects under the FLOSS umbrella, it is unthinkable reaching a person from every single community, which means that we must seek a way of making that person push the data on their own. Given those circumstances, this tracker would need to give them something in return, that is the engagement pull that originated StarFLOSS.

In this context, we proposed the StarFLOSS, a platform to observe FLOSS communities. It allows users to register FLOSS projects and set what communication platforms will be tracked, this way we are able to populate its database with information provided by the users. The platform will observe the community using the following principle “Given enough bots, all behaviour are documented and visualised”, thus we are going to focus our collectors to gather Repository, Email List and IRC channel information.

1.2 Development Methodology

At the very start of 2019, the CCSL Research Group at IME-USP submitted the proposal of a project called Ms.FLOSS to two different classes: MAC0499 – Supervised Capstone Project and MAC0413 – Advanced Topics of Object-Oriented Programming. The idea behind the double application was that the project requires a hefty amount of work,

from creating robots on the back-end to an integration layer as middle-ware to a useful dashboard on the front-end. It was a lot to require from only a couple of people. So, using the concept of divide and conquer, it was proposed to the class of MAC0413 course to help with creating a prototype as the main project of the course. That would help to ease some of the workload.

As the months progressed, the prototype developed by the MAC0413 team started diverging with our vision of what the project should be. Working with a big team lead to a loss of control on the direction of Ms.FLOSS. When it was finished we held some meetings to define the work that would be kept from that prototype and what would be remade.

The original planned architecture held a front-end layer, composed by the website and a chart generating module; an integration layer, composed by a reversed proxy and data aggregator; and a micro-services layer, composed of data collecting robots and a natural language processor. The following two paragraphs describe the changes made to it considering the prototype we had. More of the finished architecture can be seen in Section 3.2.1.

For the back-end, although the robots were in a reasonable form of work, they were not processing data, therefore we had to modify and add some new features. Still in the back-end we did not have any work done in the integration layer, thus we started to develop the data aggregator from scratch, and we changed from a reverse proxy to an API Gateway. Regarding the Natural Language processing module, it was not functioning integrated with the system. To accommodate it to our system we decided to integrate it to our integration layer.

On the front-end side, the original Ms.FLOSS had some neat features, such as user login and a graph micro-service. However, the vision of the features and uses for the platform ended up being skewed. Also, the website design was bland, its aqua green motif lacked identity. Given those reasons, we decided to restart the front-end under a different brand: StarFLOSS, that took a different visual identity approach, with a more matured idea in mind.

We specified and developed StarFLOSS in two parts. One is the integration layer, and the other is the website. As we were working on different projects and also were close, the methodology of development was more empirical, based on Agile practices. It was established a two-week sprint schedule where we would firstly discuss our objectives and then at the end of the sprint we would make another discussion to assess the progress made.

For the front-end we decided to separate the sprint in two different sections, the spikes and the programming. Spike is a term, used in agile practices, to represent an investigation period. Since we had little knowledge on all the design concepts and the tools used, they were necessary to write a better code. The programming sections were just like the name entails, it was the time where we focused on coding features.

As for the back-end, firstly we tried to write the issues to the GitLab platform, but as both members of the team had similar schedules we found more reasonable to discuss the issues and requests for data in a one-to-one meeting in the weekends before the

sprint.

1.3 Organisation of the capstone project

This work adheres to the following structure. Chapter 2 explains all previous knowledge that is necessary for following this work and explains what functionalities we planned for StarFLOSS. Chapter 3 presents about the development, what was the knowledge obtained during the process, as well as the struggles and successes that the team had while programming. Chapter 4 describes what is StarFLOSS right now, how it looks, and what it does. Finally, Chapter 5 concludes with what was achieved in relation to how it was planned and what the team suggests for the continuation of the project.

Chapter 2

Background

The main point of this chapter is to present the definition of FLOSS and the current state of software tracking. In the beginning, we introduce FLOSS, its ideals, communities, and development. We present the Git project describing why we chose it to explore the StarFLOSS features, how it started, and how its community can be tracked. We also discuss projects that have similar functionality to StarFLOSS but were not used in this project as well as the reasons they were not incorporated. Lastly, we describe the planned features and how they can bring a different experience when compared to the projects that will be mentioned in Section 2.3.

2.1 Free/Libre/Open Source Software

In summary, a FLOSS-licensed software “*means that the users have the freedom to run, copy, distribute, study, change and improve the software*”(FSF, 2007). The term Free Open Source Software or FOSS may also be used to represent what a FLOSS project is. However, it is preferred to adopt the Spanish *Libre* term as to better demonstrate that the resulting software is not free of charge, but free means that the program and its developers do not control its users. A FLOSS thus must have the four essential freedoms, quoting the original text:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it, so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this, you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

These definitions still need some clarifications to someone better understand the four pillars:

The first pillar refers to the usability of the program and not its usefulness. If the software does not have proper responses for any given inputs, it does not mean that it lost this freedom; likewise, if the program does not function in a given environment, it still has this freedom. This freedom also states that the purpose of the user should matter, not the developers. Thus anyone with any intent can use and distribute copies of the software, and the person who got that software should also share the same freedom.

The second is necessary for the freedoms 1, and 3 to have a meaning, for when one has access to the source code, they can then study and modify it, consequently redistributing to those who may concern. It is essential to point out that when a program is changed to the point that another user cannot run in their environment, a situation called "lock-down", this freedom is no longer achieved, and the software can not be said free.

Freedoms 3 and 4 say that one has the right to distribute their software in any way they want, either for free or charging something; in the last case, the resulting software would not be granted a FLOSS license.

The StarFLOSS project wants to observe how FLOSS communities are structured and how they work. To have the understanding of how we are going to achieve that we firstly describe how the social structure of the project is. FLOSS projects may or may not have a dictatorship where the "owner" of the project is a benevolent dictator. The fact that any FLOSS project has "Forkability", which is the possibility to have a complete copy of the project in their repository, makes any controversial decision of the dictator of the project a suicidal venture, thus delicate issues are always discussed in the available channels of the community.

As the community takes a big part in the decision making process and the developers that are responsible for the maintenance of the project are distributed around the globe, a FLOSS must have a couple of channels available for efficient communication. Usually, the following are available:

- a Website for developers, users, and other actors to participate in the project
- a Mail List as its main channel of communication and record of the discussions that led to patches and changes in the project
- a Version Control repository to allow participants of the project to be in touch with the project quickly
- a Bug tracker which is not only for finding bugs, but to request new features, one-time tasks, and generally any change that has an initial state, end state and a period to transition through the states
- a Real-Time Chat System is also significant in a project, for it allows its users to have instant responses about critical issues or what would be considered silly questions in the Mail List; usually, the tool used is IRC(Internet Relay Channel)
- Wikis are a great tool, as they keep all relevant information in one centralised place, they help new members to understand the scope and function of the project and function as a way to notify developers when there is a change.

From this list, we pinpointed where are the places we ideally sought to observe. Initially we decided to use information from the Git Repository, since Git is a version control system, there is no change in a project code that is not committed on it. We would have a reliable history of the entire project life. However many important decisions are not made in commits, thus we also choose to track Mail Lists to have the conversations that led to commits. Lastly we also decided to use the IRC channels, since the data would be informal conversations from people who started on the project to long-term supporters, it would provide us with information about the interactions between these two groups.

2.2 Git

We selected the Git project as the example of use to explore the StarFLOSS feature in this work. Git is a distributed Version Control System (VCS), widely used for its simplicity and efficiency, it has impacted all interactions of FLOSS communities.

Anyone can build a VCS platform such as GitHub or Gitlab thanks to the Git project, using it to track changes in the source code during software development. The creator of Git is Linus Torvalds, the same as the Linux Kernel. During the development of the Linux, since 1991, the source code was distributed and tracked by patches and archived files. In 2002, the Linux Kernel project started to use a proprietary VCS called BitKeeper¹, which was free-of-charge back then. Using proprietary software to track a flagship FLOSS project raised many concerns in the community, and in 2005 the status of free-of-charge was revoked. The change in their status made the community stop using it, and as a result, Linus and the community decided to create a new project to at last solve the problem of tracking and sharing code – the Git project. The foundation of the project lies in principles such as:

- Simplicity
- Speed
- Ability to handle extensive projects efficiently
- Fully distributed
- Supports non-linear development (branch system)

To become a Git project contributor, a newcomer must write their patches and send it through an email list, similar to the Linux Kernel project. Beginners should subscribe to the email lists available in the Git website². Not only sending the patches via email list, but the issue tracker can also be found in the lists too. The other option of contact would be via IRC (Internet Relay Chat) channel “git-devel” hosted in the “irc.freenode.net” server. IRC is an application layer protocol that eases communication through text and is used in the community when a more personal discussion is needed. Thus the main focus in gathering data of this project would be these three channels of communication:

¹<https://www.bitkeeper.org/index.html>

²<https://git-scm.com/community>

- Email lists
- IRC channel “git-devel”
- Git repository of the project

At the University of São Paulo, the FLUSP (FLOSS at USP)³ group helps beginners learn how to contribute to FLOSS projects. Our proximity to the FLUSP members makes it more feasible to verify information quickly. Moreover, the size and the age of the project are other reasons to select it as the example of use. Being a relatively young project, we can have the majority of the lifespan of the Git project trackable, and having about sixty thousand commits means we have the right size to start our services.

2.3 Related Projects

We investigated some software analytics projects to have a better knowledge about this area and to find projects closely related to our proposal. Software analytics brings the developers some useful insight into how software is done, how teams coordinate, what are the best practices according to metrics, not the subject experience of developers. The data gathering is around the software artefacts, as well as the related processes of their producing and evolution. Having in mind the proposal of StarFLOSS, we present two projects as references to our platform.

The first project is called OSS Browser, which guided an ethnography study by [DUCH-NEAUT, 2006](#). Its architecture is simple; it has a back-end made in Pearl that retrieves two categories of data – Emails and CVS (Concurrent Version System) logs. The data is then processed in the back-end and displayed in a layout made in a Java Applet that can be viewed in any web browser. Figure 2.1 is a sample of the visualisation of the software. The OSS Browser project has a very compelling interface with many interactive elements and, even with only one graph, can show a lot of information. It owns many features that StarFLOSS strives to achieve.

Another project is named Perceval, which is a system, written in Python, to gather data from multiple sources consistently ([JESUS M. GONZALEZ-BARAHONA and COSENTINO, 2018](#)). It works as a service that uses a command-line tool to issue due tasks. Percival is part of the project called “Grimoire Labs”, which performs as a stack of services to provide visualisations of software development data with Kibana as the tool to generate the visualisations. Kibana is a popular data visualisation plugin for the Elastic Search project. It allows the quick creation of powerful dashboards with log analysis features, machine learning, and many different kinds of charts.

Regarding the front-end, since the stack of Grimoire Labs uses Percival service, we would have to use Kibana in our project. However, Kibana was built to be a DevOps tool, not an overall chart creator. Even if it is easy to create great charts with it, Kibana does not offer robust customisation tools. It also does not provide natural ways to extract the

³<https://flusp.ime.usp.br/>

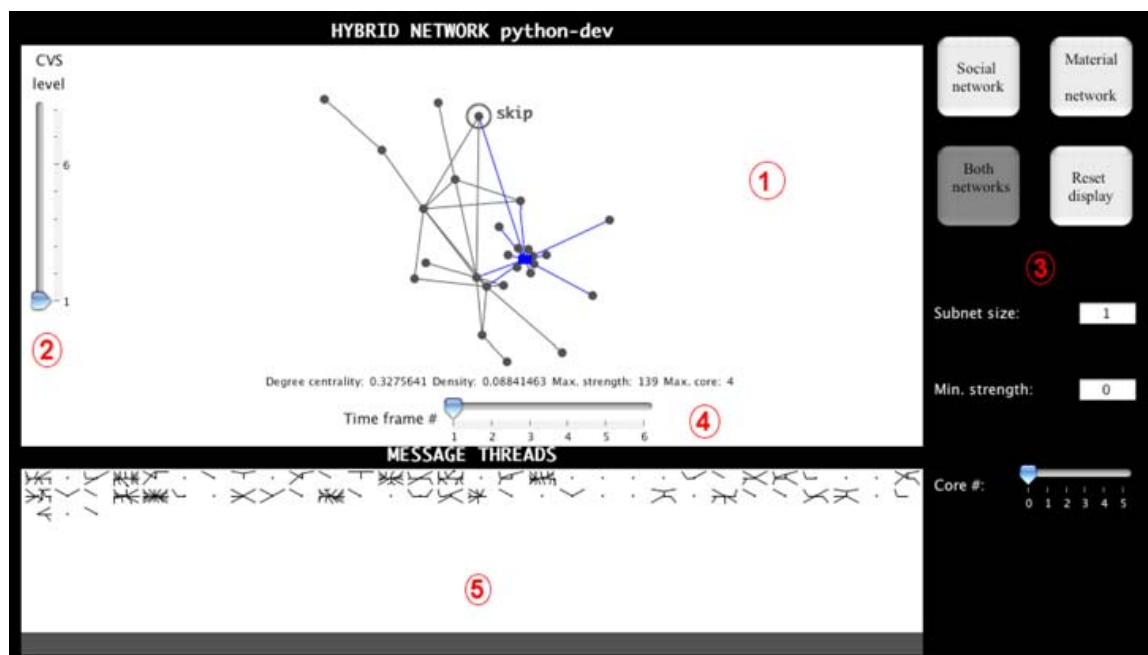


Figure 2.1: Example of visualisation of the software used in *DUCHENEAUT, 2006*

graphs provided on its platform. This issue would limit the possibilities of what StarFLOSS could do both concerning user interaction and artistic direction.

Regarding the back-end, we chose not to use Perceval as it is a service for collecting data that does not store data or process it. Furthermore, it does not work as a micro-service, as it would need a database outside of the service. Another point is its many responsibilities. The service is indeed a complete solution to collect data. Still, as it has many responsibilities, it is once again out of a micro-service approach, and that would not be beneficial to our proposed architecture. Besides that since we have our base services and our Integration layer planned to be done or partially done in Ruby, having another language to the project would carry more time spent in adaptation to other language. Listing all these reasons we decided to implement our collectors as we were planning, with ruby standalone micro-services

2.4 StarFLOSS features

To be distinguished from the previously mentioned projects, we designed StarFLOSS to be easy to use. Our aim is that any user who lands on the website can understand the data, whether they know a community or not. Taking clues from sites such as “Visualising MBTA Data”⁴ and “Out of Sight, Out of mind”⁵, our goal is to be an easy to use, engaging take on the subject. However, StarFLOSS started as a scientific tool that should help demonstrate the state of a community for any interested researcher. Therefore, we planned the back-end in a way that someone could use the data with a more focused front-end.

⁴<http://mbtaviz.github.io/>

⁵<https://drones.pitchinteractive.com/>

2.4.1 Front-end features

The primary purpose of StarFLOSS is to be a display of FLOSS data, an easy to understand visualisation. Thus, anyone interested in a specific community could better comprehend its activities status. We designed all StarFLOSS features thinking about what would better help the user find crucial information. There are two different ways in which we could help: first, in how to find the project, and then in how to find the data.

We need to meet some requirements to create a good user experience. For example, if a user knows what project they are looking for, they should have no issue finding it; otherwise, if they want to see the project that has the highest number of commits, that should be easy as well. The platform should have a search bar and relevant search parameters in order to give that power to the user. As a way to also provide visibility to smaller lesser known projects, it was decided that the home page of the website would display some random projects, that is a fun surprise to the user and a quick way to present projects that would be less searched for.

Concerning the dashboard, the best way to generate interest is to allow the user to interact with the graphs directly. By presenting options for the user to filter the data they want, it is possible to create a more personalised experience, this filtering creates a more memorable user experience, according to [BILL SHANDER, 2016](#). We followed the Shander's approach to design the StarFLOSS dashboard. It is not to be packed full of different charts and data, but to have a few well-developed charts, to provide the best experience for the users.

Create a fun visual identity for the StarFLOSS was not the main focus of this project, but not following the rules of visual design generates an unappealing website, becoming a barrier to attracting users. Data visualisation inherently comes with the need for good design. A dashboard with no cohesion is unpleasing to look at and can be hard to read (as discussed in Section 3.1.2). StartFLOSS visual design took a lot of planning.

2.4.2 Back-end features

We designed the integration layer with the following guidelines in mind: function as a "Load Balancer" for the incoming requests, so the system is not easily crashed; making it seem like the client is requesting the data from a single service, hiding any further complexity; aggregate incoming data to more specific requests; and also be a service that functions independently, the information should be easily consumed by any eventual service that requests it. Using these guidelines we would like to present some features by the end of this project:

- Any user can add projects to the platform given they are FLOSS and have public repositories
- Provide the history of commits of a given project
- Process statistics of the commits
- History of commits of a specific user in a repository

- Filter raw information from the bots and provide refined information to the platform
- Standard API documentation
- Provide a tracking tool for messages in the IRC community
- Provide statistics of the IRC channel related to a FLOSS
- Analyse sentiment of messages in the IRC channels
- Give the user the possibility to upload a historical of data sources that do not have a pattern as archived email lists.
- Allow other developers to contribute to the Integration Layer quickly
- Use of Continuous Integration in the build, test, and deploy stages.

With these planned features we understand that we may achieve our goals presented in the beginning of this section. With that a very flexible platform will be created, pleasing casual users interested in discovering new information in a beautiful website, to a more focused profile intending to analyse what it is provided. Having described our planned goals, in the next chapter, we focus on the development process that created our system.

Chapter 3

The development of an Observatory of FLOSS Communities

This chapter describes how we developed the platform and what became of it. We present the implemented features of both the Integration Layer and the User Interface, describing the technologies and the external dependencies used in each of them.

3.1 The user Interface

The StarFLOSS website is the first thing the user sees when discovering this project. With that in mind, we explain in this section the idealisation process, the design fundamentals, and how we applied them to the project. Moreover, we also present the coding process and the choices that had to be made to create a great user experience. We discuss topics from the many different disciplines that were necessary to study to make StarFLOSS, from typography to data flow.

3.1.1 Creating a visual identity

The phrase that guided the visual graphics development of StarFLOSS originated from the subtitle “an observatory for FLOSS”. We decided to take that metaphor to heart and base our site around this star concept; this way, we give a visual aid to the user for better comparing the software projects. A real-life observatory is “a building equipped for studying the planets and the stars”¹. Given that, it is not hard to figure that to give the metaphor continuity, the projects should be celestial bodies.

We decided that the projects would be stars, because of the ease of understanding in comparing stars. Looking at the sky to see different stars is something that the average

¹Definition taken from the Cambridge Dictionary

user is sure to have done. While comparing planets, asteroids, or galaxies would be a bit more challenging. Also, stars are a better metaphor since we already use it in our language. A bright star is good or thriving. A dying star is dim and small. There could be a case for thriving and dying planets, but the visual representation of a thriving planet differs quite a bit from person to person.

A negative point for stars is that there is little visual distinction to work with. This could be an argument for using planets on the metaphor. Things such as flora, atmosphere and even the composition of the planet, if it is gaseous or a rocky, are very distinct characteristics that planets hold. A star is just a ball of flaming gas held together by gravity. From far away though, most things end up looking homogeneous. A distant planet is visually just a star, none of its main characteristics shine, so this point does not hold well. To differentiate stars, we can change colour, size, and brightness; in real life, of course, those are all related. However, we are going to have to take some poetic licenses to make the visualisation work.

After deciding what was going to be the project's visual identity, a brainstorm to make the StarFLOSS storyboard (Figure 3.1) was done. We created it using paper and pencil because a storyboard should be an easy to modify diagram of what the website will eventually look like so that the team can have the same overall idea in mind while programming it.

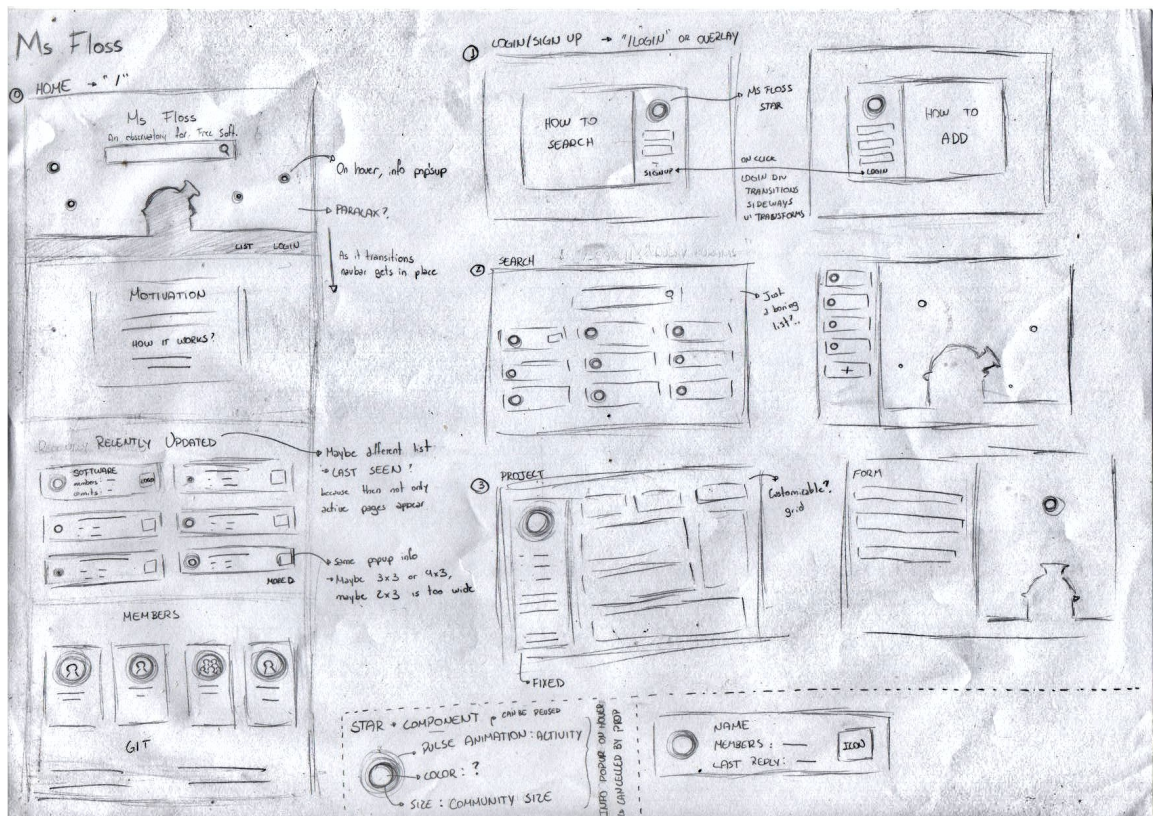


Figure 3.1: StarFLOSS storyboard

3.1.2 The importance of visual design fundamentals

Visual design is not a widely approached topic in the realm of computer science. For someone learning computer science, the text may appear on a Human-Computer Interaction book, while the content will most likely be a digested version of what encompasses visual design. Regardless, knowing the subject is essential for front-end developers. Brad Frost² says it himself in his blog that HTML and CSS development is also design, and that the most rewarding projects to work, are the ones that understand that. For this reason, people who do not understand why front-end development needs to be involved in the design process from the beginning are often inclined to be frustrated with the final results of a project (BRAD FROST, 2013).

For data visualisation, the use of charts and maps to display visual information is one that precedes computers by a long shot. Michael Friendly studied the origins of data visualisation, in his text “A Brief History of Data Visualization” (MICHAEL FRIENDLY, 2006), he explains that data visualisation started appearing in history with geometric diagrams, in tables with star positions and other celestial bodies, and the navigation and exploration maps. Later, the Ancient Egyptian surveyors used the idea of coordinates in laying out towns. And, the location of the earth and the sky were positioned using something similar to latitude and longitude at least by 200 BC. Being such an antique topic shows how visual design is such a vital knowledge in data visualisation and its importance in a project such as StarFLOSS.

So, what exactly are the fundamentals of visual design? This question is hard to answer; most designers, when asked, will say very different things, as what makes someone perceive a view as pleasing is very personal. However, there are some common topics that will appear, such as contrast, balance, emphasis, movement, white space, proportion, hierarchy, repetition, and many more. There is no definitive structure on how to group those guides, so for this section, we will use the separation Tony Harmer uses in his course “Introduction to Graphic Design” (TONY HARMER, 2018). To him, three big categories need to be taken into consideration to make a good design: the Layout, the Typography, and the Colours.

Layout

The basics of layout tend to encompass alignment, padding, proximity, repetition, a lot of things that are very explicit when using HTML and CSS. So it tends to be the most common knowledge to a programmer. Some more advanced topics focus on the realm of composition, balance, and hierarchy.

On StarFLOSS, most of the composition relies on calling attention to contrast points. The night sky is a very dark colour, so any white point against it creates a contrast point. Contrast points give a component a higher hierarchy. As you can see in the StarFLOSS homepage (Figure 3.2), the title calls a lot of attention and, below it, we have a black observatory against a brighter section of the image.

²Brad Frost is the author of the book “Atomic Design”, which introduces a methodology to create and maintain effective design systems.

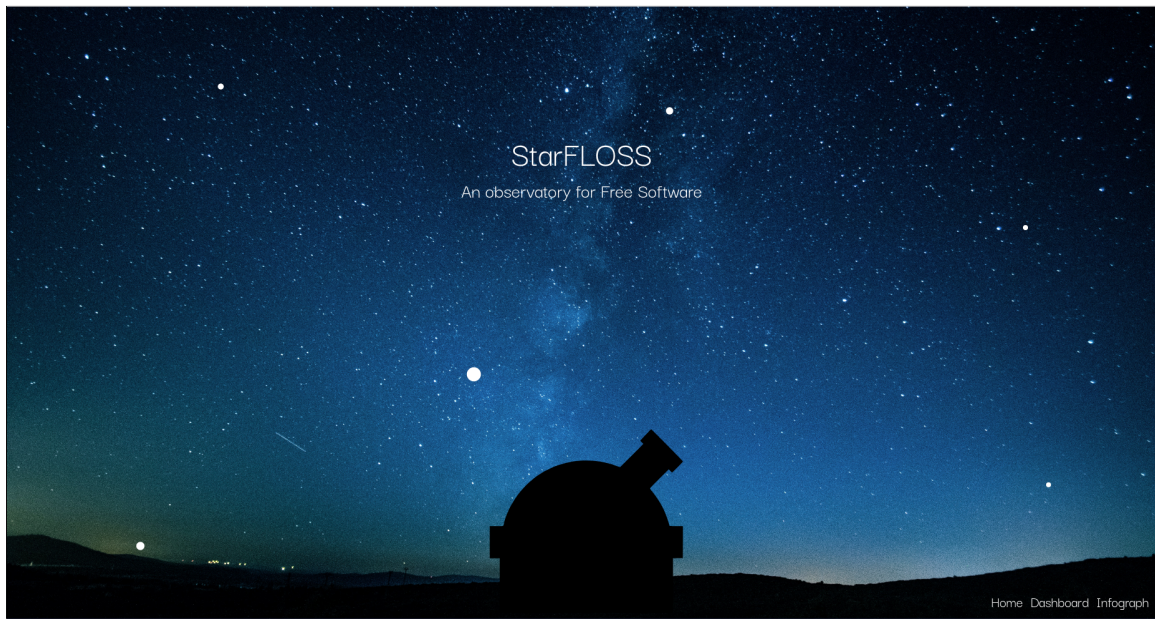


Figure 3.2: *StarFLOSS home page*

Typography

Typography tends to be a less discussed topic compared to the other two. Most likely because it is the only subject that does not accurately translate to visual arts, nevertheless it is an important one. Ina Saltz starts her course “Graphic Design Foundations: Typography” (INA SALTZ, 2013) with a chapter titled “Why good typography matters”, and states that Type can help users to form a critical first impression about your message; thus, it is essential because it causes an impact to the user even before it starts to read it.

For StarFLOSS, none of the team knows a lot about typography, so the font choice was a very safe one. The website has only one font, to not risk having a clashing combination of them, and the chosen one is the Darker Grotesque (Figure 3.3). The Darker Grotesque is a grotesque, geometric font that, like many other geometric typefaces, gives the website a more modern, futuristic look, especially when compared to the neo-grotesque types³ commonly used on the web. Neo-grotesque typefaces are made to be super readable, which makes them excellent body text fonts. In contrast, when using a geometric typeface, some readability is lost. The single-story lower-case “a” in the Darker Grotesque, one of the critical features of a geometric typeface, makes it less distinguishable from letters like lower-case “o”, as does the roundness of the lower-case “c”. The StarFLOSS, however, is not text-heavy, and Darker Grotesque does bring it better suiting aesthetic, and as such, it is the used font.

³Examples of neo-grotesque typefaces includes Arial, Helvetica, MS Sans Serif and Univers

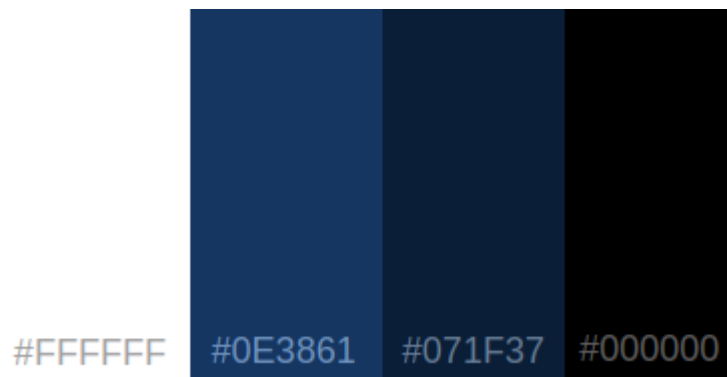


Figure 3.5: *Blue colour palette*

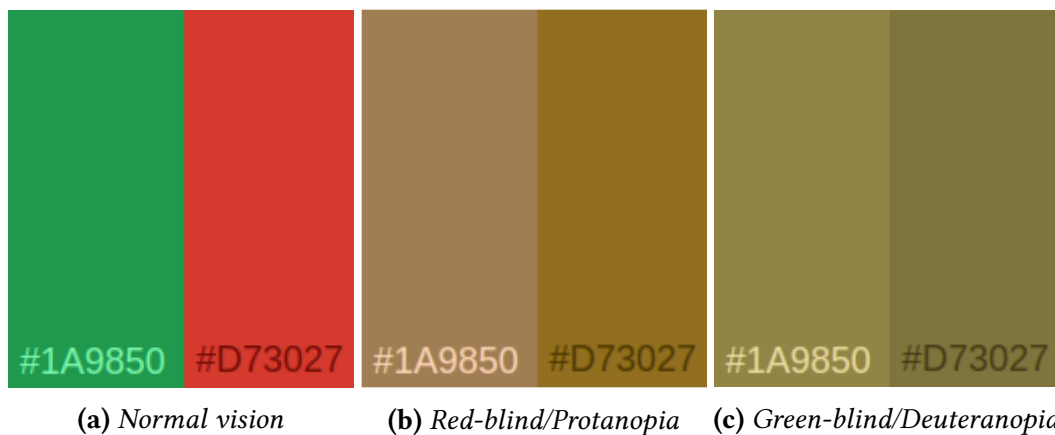


Figure 3.6: *Red and green colours as seen by colourblind filters*

3.1.3 A solid website with React

Leaving the design work to focus more on the programming, we now start talking about the project decisions of StarFLOSS.

StarFLOSS is made by using the React framework. React is a trendy javascript library for building user interfaces. It was developed and is used by Facebook and has been growing more and more in those past years. One of the main advantages of React is how easy it allows developers to re-utilise code. It has a component system that makes each part of the front-end its own compartmentalised feature and allows for easy reuse of code. It has been securing its place as one of the best front-end frameworks to use⁴ and, for that reason, it was chosen as StarFLOSS's framework.

Learning React is not hard. The documentation is excellent, and tutorials are abundant. The issue with it arose when looking at the different libraries that were needed to be used in conjunction to complement the library. React is an excellent library for creating Single

⁴A quick read, if you want to see research into this, is the "Tech Trends Showdown: React vs Angular vs Vue" article. <https://medium.com/zerotomastery/tech-trends-showdown-react-vs-angular-vs-vue-61ffaf1d8706>

Page Applications (SPA), but it does not stray from that objective. To have some extra features, we need to go after other open-source projects, that may not be as well supported as React itself.

Two libraries gave the team some issues were “react-router” and “react-redux”.

“react-router” is a library for allowing React to use URL routes. As it is not a requirement for SPAs to use different URLs, nor to deal with browser history, the library fills this need. The problem we had with the react-router was the documentation. The official documentation comes in the form of a tutorial, granted it is a very well structured tutorial. However, sometimes it lacks information on how a component works, then you can only rely on the community for help.

“react-redux” is a fantastic API that helps implement the Flux pattern on a react project. Flux is an application architecture created by Facebook to make websites have a single source of truth. It is a pattern that challenges the conventional Model-View-Controller (MVC) to have a unidirectional data flow; the pattern in itself is more discussed in the appendix A. The concept of Flux is straightforward; however, the Redux API is complex⁵. Different components must follow various rules, and it does add a lot of complexity to the code. Currently, StarFLOSS only uses Redux for managing the list of available projects. However, if the dashboard could have a system like that in place it would facilitate some functionalities; a more in-depth discussion of betterment for the website can be found in Section 5.1.1.

Another decision we made when developing the front-end was that no pre-made component would be used, even if using React facilitates this reuse of code. In other words, all of the components were made from scratch, only using HTML tags and CSS. This decision was a request of the developer as they wanted to try to experience the whole website creation. Coding a web-page that strictly follows a given design is a vital skill for a front-end developer, so they decided to hone that skill, even if it meant adding overhead when developing.

3.1.4 An interactive dashboard with D3.js

We chose D3.js as the library used to create panel charts for the same reason we selected React as a framework: it is the most used library when talking about data visualisation.⁶ D3 is very popular because it offers the best customising possibilities. It is not a chart generator library, but a data manipulator one. This means that D3 does not have any code that can draw a bar chart, but it has code that aids the programmer on drawing each rectangle that will eventually compose the chart. It allows the programmer to do any visualisation, animation, and interaction that comes to mind, without having to sacrifice their artistic vision to a pre-made mould – given that they have the coding knowledge to do that.

⁵This Hackernoon post accurately displays how one feels when learning Redux: <https://hackernoon.com/how-i-felt-while-learning-redux-de16fb2f5ad2>

⁶Only comparing Github stars, D3.js has 88.5k, while chart.js, a very popular competitor has 46.1k.

The most prominent problem of D3 comes with its best advantage: because of all the freedom it offers, it is very complex to use the library. D3 is known to have a steep learning curve, the reason why it is a very discussed topic in the community. However, the fact that it is hard to learn is a fact accepted by all⁷.

Now, that we have presented the reasons why to use D3.js and the issues that came with that choice, we should discuss what exactly is D3.js. In his article “D3 is not a Data Visualisation Library”, Elijah Meeks gives a great visualisation of the D3 API (Figure 3.7). He describes it as such “A hierarchical diagram of the functions listed in the D3 API page⁸, grouped into their category (such as d3-scale or d3-array) and subcategory (if applicable, such as continuous scales) and then further grouped and coloured and labelled by the part of the API they represent. In this formulation, the geospatial data visualisation functionality is a subsection of DataViz.” (ELIJAH MEEKS, 2018).

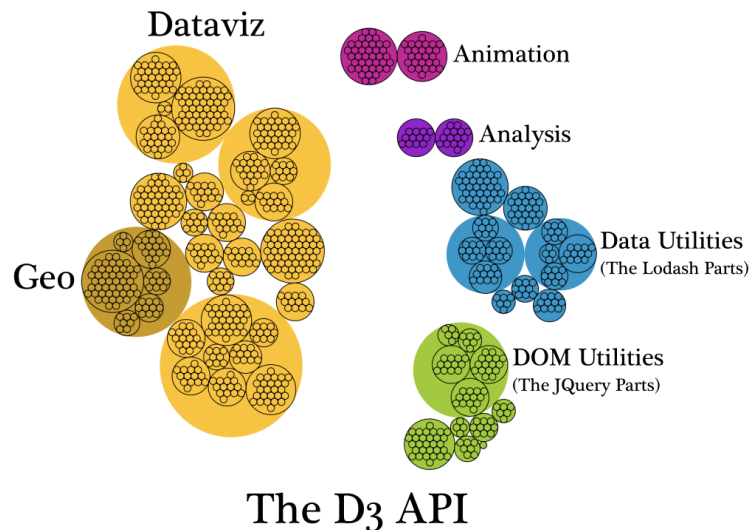


Figure 3.7: D3.js, by Elijah Meeks

By looking at the D3 visualisation (Figure 3.7), one can see how vast the D3 API is and for what it is geared. Every category holds functions that are necessary for drawing a chart with D3:

- *Analysis holds libraries such as d3-DSV and d3-quadtrees. They are used to read data from a formatted file, such as CSV.*
- *Data Utilities holds libraries such as d3-time-format and d3-interpolate. They help with processing data and extracting information that may be necessary to draw the chart, such as getting the max value for a chart legend or grouping data by name.*
- *Dataviz holds libraries such as d3-colour-schemes and d3-shape. Those are responsible for helping create SVG shapes in the canvas, such as drawing complex paths and drawing common chart parts, like scales.*

⁷“The trouble with D3 is to build a visualisation you must also have a deep understanding of SVG, DOM, JavaScript, geometry, colour spaces, data structures, the standard model, and quantum physics” - Martin Bunch, on twitter

⁸<https://github.com/d3/d3/blob/master/API.md>

- *DOM Utilities holds libraries such as d3-selection and d3-zoom.* They help with integrating the chart with the rest of the DOM, such as handling events when the user decides to drag part of the chart or showing a tooltip when they hover over a rectangle.
- *Animation holds libraries such as d3-ease and d3-transition.* They create the animations when the data changes, they are not that necessary if you want a simple static graph, but they can make a chart pop if used correctly.

Those categories show every step necessary to draw a chart in D3. By using D3noobs code⁹ as an example (Figure 3.8), it is possible to have an overall idea of how to create a D3 chart:

⁹The original code can be found at <http://bl.ocks.org/d3noob/4414436>

```

<script>

// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%d-%b-%y").parse;

// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

// Define the line
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Add the valueline path.
    svg.append("path")
        .attr("class", "line")
        .attr("d", valueline(data));

    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    // Add the Y Axis
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

});

</script>

```

Figure 3.8: Code to generate a simple line chart (Figure 3.9). In this image we colour the D3 code lines following the scheme in Figure 3.7, the red lines are SVG code and the non coloured are plain Javascript.

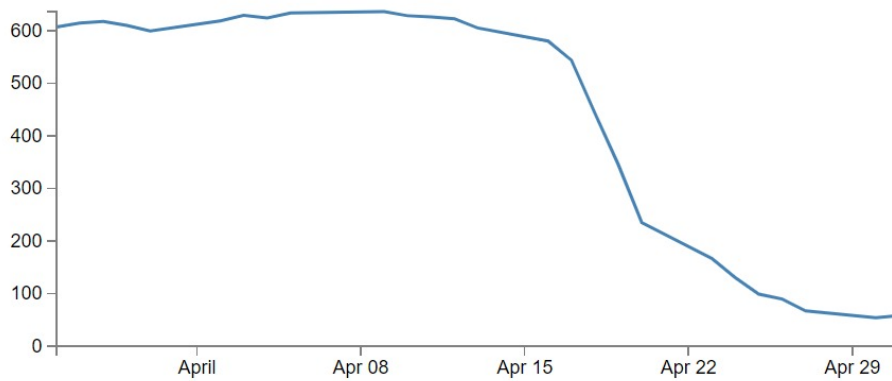


Figure 3.9: Line chart generated by the code in Figure 3.8

StarFLOSS charts have a lot of additional features than the ones in Figure 3.8, such as added animations and user integration, but the overall basis remains the same. The main difference is that the data that feeds StarFLOSS is not stored in a file, but retrieved by consuming the API provided in the Integration Layer.

3.2 The Integration Layer

When the project was idealised, it was clear that, in the future, it would have multiple micro-services serving the front-end of the platform. Although the idea of having many standalone services seems quite robust, if we left it at that, the front-end would have the additional tasks of aggregating and refining the data. Thus we built the Integration layer to be the Aggregator of the system. Besides that some principles were also applied to be in harmony with the rest of the architecture:

- *Separate data storage:* Although the Gateway has its database to store selected information for a project, each micro-service has its own data storage allowing the system to have Polyglot-Persistence a term used by Martin Fowler¹⁰ to describe the usage of different Databases throughout a system.
- *Independent deployment:* The integration layer does not depend on any micro-service to be deployed.
- *Everything documented:* Every bridge presented by the Gateway must be documented on the homepage of the Gateway. It facilitates the use by the front-end and anyone interested in the collected data.
- *Free and open source Software:* We must develop the Gateway in a way that eases contribution. It must be comfortable and smooth for project maintainability in the future.

¹⁰<https://martinfowler.com/bliki/PolyglotPersistence.html>

3.2.1 Architecture

In this subsection, we describe the decision-making process of the architecture. Our first option was to build the entire system in a monolithic architecture. It could make the development of the platform simpler because all the systems shared the same database. There would also be a single pipeline of deployment, and everything would be in a single code base. However, this option entails some limitations that would not be beneficial for a long term project. As an example, a single database throughout the system also means a single point of failure to the entire platform.

The Integration Layer aims to provide a middleware infrastructure that can scale and evolve to adapt to possible variations of requirements. To achieve this, we decided to follow a Service Oriented Architecture with principles of micro-service Architecture.

A Service-Oriented Architecture (SOA) is an umbrella term for principles that facilitates integration among systems. SOA was firstly used by the Gartner Group (YEFIM NATIS, 1996). It is defined as a design paradigm that helps systems meet business demands, reduce costs, increment the consistency and agility of the application, and produces inter-operable, modular systems, called services, that are easier to use and maintain (DRAGONI, 2016). To see the real benefits we have using this paradigm, we present a comparative description of these two styles:

- **Scalability:** the monolithic approach may limit the system from scaling when there is an increase in load for a specific module; in that case, the entire platform may be compromised. In contrast, the SOA approach dilutes this load into the services. It results in a more reliable system, as it does not depend on any service, and the platform then scales healthily.
- **Maintainability:** due to its vast code base and the equally massive complexity of the system, the monolithic style does not age well. SOA, on the other hand, has services that can evolve separately. Therefore, if any service starts being a problem for the developers, it can be quickly rebuilt without noise to the system.
- **Deployment:** a single change to the code in the monolithic approach may cause the need for an entire platform to be deployed again, causing possible losses to the owner while it has been out. The SOA approach has each service having its pipeline; therefore, there is no need for a full reboot.

Following those ideas, we developed the architecture depicted in Figure 3.10. As we described before in Section 1.2, some changes were made to it, compared to the first iteration. The Integration Layer is represented by the API Gateway and Data Aggregator, which has the responsibility to handle the requests made to the services and filter the information according to the client. The Collectors in the picture represent the layer of micro-services used to collect and provide information from the Data sources. Their basic principles are independent of vendors, products, and technologies. They are self-contained units, the client knows nothing about its complexity and they can exchange messages with other services at will.

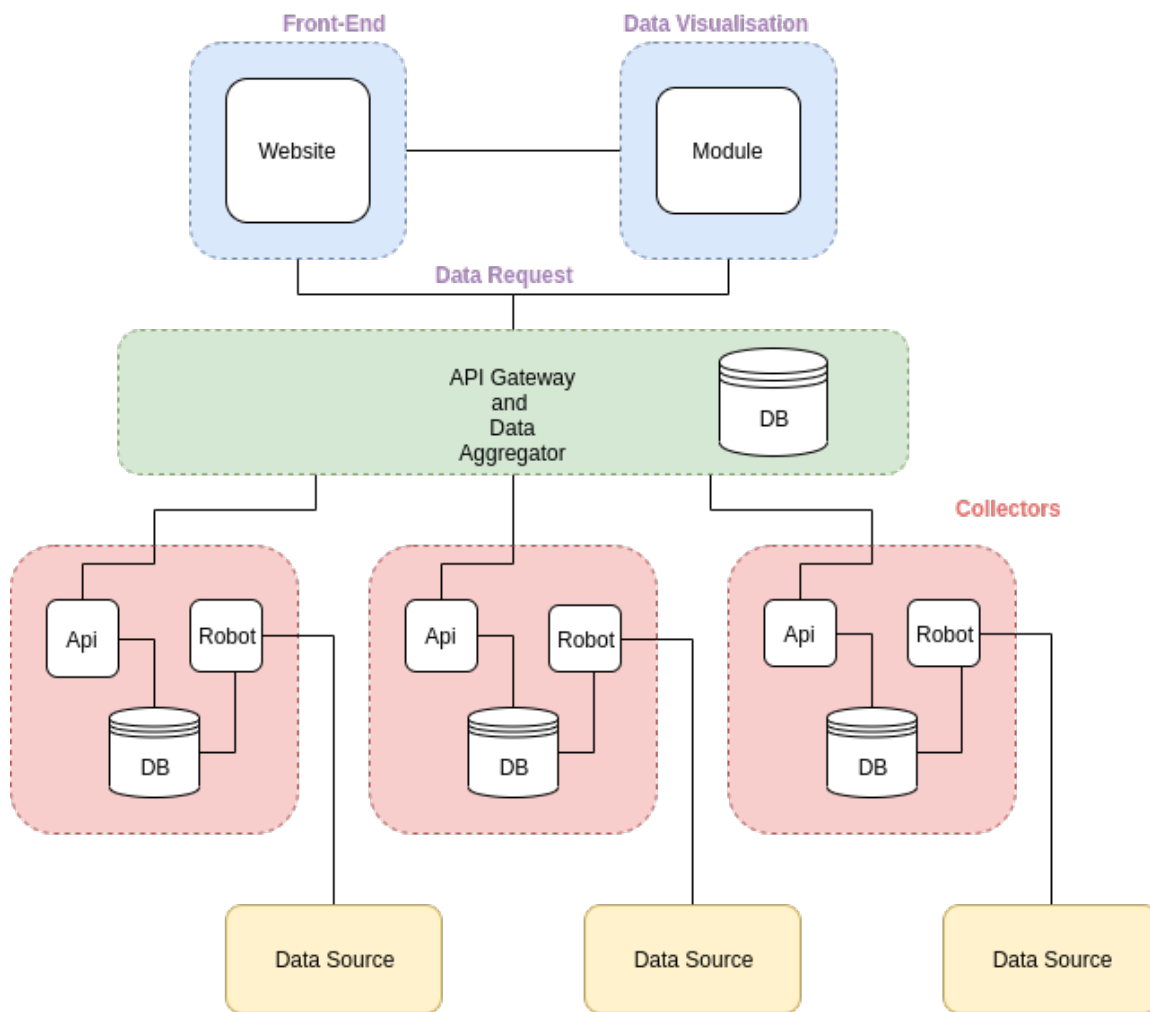


Figure 3.10: *New Proposed architecture*

With this approach, we may achieve the values needed for the continuity of the platform, stated in the SOA manifesto¹¹:

- Business value over technical strategy
- Strategic goals over project-specific benefits
- Intrinsic interoperability over custom integration
- Shared services over specific-purpose implementations
- Flexibility over optimisation
- Evolutionary refinement over pursuit of initial perfection

¹¹<http://www.soa-manifesto.org/>

3.2.2 Good Practices

Usually, in traditional software development environments, when a member or a delivery team is due to launch its code to the production environment, rarely this process is done with ease. Hours or even days are spent in the pursuit of integrating the system, hence why this moment received the popular name of “Integration Hell”. A good practice to avoid those issues is implementing Continuous Integration (CI) in the project. With CI, delivery teams, and active members of the project, when submitting their codes, have automated builds to test its integration with the current head of the project. In this StarFLOSS back-end, this is done using the Git-repository manager **GitLab**.

In this project we are using the features of GitLab for Continuous Integration and Deployment (CI and CD respectively). When a member makes a “Merge Request”, and it is accepted, their code is going to pass through three stages before being added to the master:

- **Build:** The script tests whether or not the project is building without fail.
- **Test:** All the test scenarios are running to catch any possible bugs. This process is also portrayed as Sanity tests using Unity and integrated tests.
- **Deployment:** The project is deployed in a virtual machine in the cloud using the platform Heroku¹² if the integration is successful, the Gateway is ready for usage.

Heroku is a famous tool used for deploying applications. In our case, the tool was great for continuous deployment and to test the platforms. However, this environment proved problematic when used with the micro-services. For example, the Commit bot has a version deployed to a Heroku app instance. This application runs in a lightweight Linux container called Dynos. Dynos have what is termed Ephemeral Filesystems. It means that, on average, once a day, the Dyno is replaced by another one, replacing its filesystem for a new one. As the commit bot uses an SQLite database, whenever Dyno is rebooted, the service loses all its data. For this reason, we decided to use a virtual machine hosted in the University to deploy the micro-services.

Besides the CI and CD practice in this project, we also used the readme.md file in the repository to give all the guidelines from installation to contributions. This way, not only students can contribute, but any programmer willing to help out.

After its start, we also added to the project another good practice to ease contribution. Initially, the commits were sparse, and many changes were made in just one commit. It caused difficulty in locating bugs introduced to the project and regression of the software in case of need was practically impossible. During the contact with free software projects, we noticed a pattern in the commit messages and in further studying of this pattern the Seven Rules to write a good commit was found¹³. The quality of the commits has increased using those rules, and debugging or finding the new features introduced became an easy task.

¹²<https://www.heroku.com/>

¹³<https://chris.beams.io/posts/git-commit/>

3.2.3 Restful API

To comply with all requests done to the service, we decided to use a software architectural style called **REST**(Representational state transfer). This style is used in most web services and allows the integration layer to achieve:

- a Client-Server Architecture, allowing both front-end and back-end to evolve independently;
- statelessness, so that all requests must be free from client context;
- cacheability, allowing clients and intermediates to cache responses for further scalability;
- a Layered system, which means that a client does not need to know if they are communicating with the actual server or an intermediate;
- a Uniform interface to simplify and decouple the system for even more independence.

3.2.4 Ruby on Rails

The framework chosen to build the Gateway was Ruby on Rails. Quoting its definition, it is “a FLOSS web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern.”¹⁴. Describing the MVC pattern is essential to understand what we utilise and do not from this pattern.

The MVC pattern is a guideline; it separates the project into three sections:

- The Model Layer represents our domain of the database. For instance, the Email model represents the business logic we have for this kind of data and encapsulates it so we can use it as an object for further refinement.
- The Controller Layer is responsible for handling the incoming requests. The action dispatcher routes to the appropriate controller and then render a response, in a view or a custom format. In our project this is very often represented in a json format.
- The View Layer is composed of many templates used to visually represent the API resources.

In this project’s Integration Layer, we decided not to use the View Layer as it would not be responsible for the representation of data. We used, therefore, the Rails 5 API only application, removing unused middle-ware like Action View, and generating a faster and lightweight application. We used the controller layer for aggregating all methods related to each of the data sources; for example, we have a controller for the commits data source, one for the Email data source, and so on. The Model Layer was used primarily for storage of Projects data; that is, each project should have an entry in our database telling us the necessary information.

¹⁴<https://rubyonrails.org/>

The **Project Model** is the primary data object of the Gateway. It is used to store information about the other services and how to reach out to them. As it is shown in the picture below, it has (1) one “id” to identify the object in the Gateway, a “project_name” to be presented in the front-end, a “git_url” and a “git_id” for a representation of the object in the commit bot micro-service. It also has “email_list” and “email id” for a description of the email list micro-service and “IRC channel”, “IRC server” for the representation of the IRC micro-service. Lastly, two timestamps hold the date of project creation and its last update.

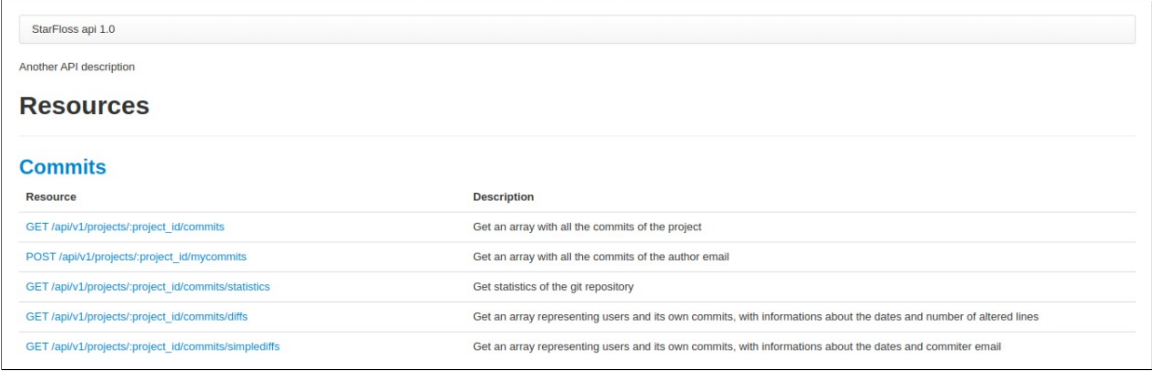
```
1  {
2    "id": 980190962,
3    "project_name": "Testing",
4    "git_url": "url.test",
5    "git_id": 3,
6    "email_list": "git",
7    "email_id": 1,
8    "irc_channel": "irc.freenode.net",
9    "irc_server": "my-channel",
10   "created_at": "2019-10-19T23:12:02.008Z",
11   "updated_at": "2019-10-19T23:12:02.446Z"
12 }
```

3.2.5 Documentation

A good API must have proper documentation to make it easy for any user, even if it is not a contributor to the project, to consume its endpoints. There are several suitable solutions for this problem, and the choice for this project was the library “*api pie-rails*”¹⁵. *Api pie-rails* is described as a Rails engine for documentation of Restful API without the usage of traditional comments in the code; an example is found in figure 3.11. This library allows documentation with ruby code bringing the following advantages to the project:

- No other syntax is needed to learn, as it is made with ruby code.
- The code can be used for validation
- Documentation is available using the router of the application
- Documentation is made automatically if tests are written
- The pages are user-friendly and intuitive.

¹⁵<https://github.com/Api pie/apipie-rails>



| Resource | Description |
|--|--|
| GET /api/v1/projects/project_id/commits | Get an array with all the commits of the project |
| POST /api/v1/projects/project_id/mycommits | Get an array with all the commits of the author email |
| GET /api/v1/projects/project_id/commits/statistics | Get statistics of the git repository |
| GET /api/v1/projects/project_id/commits/diffs | Get an array representing users and its own commits, with informations about the dates and number of altered lines |
| GET /api/v1/projects/project_id/commits/simplifiediffs | Get an array representing users and its own commits, with informations about the dates and committer email |

Figure 3.11: *Example of documentation page in the application*

3.2.6 The Commit Bot

The platform will be composed of micro-services to provide the information needed. The first one we are going to describe is the Commit bot. Its main task is to keep track of the repositories of FLOSS projects. It completes this task by cloning the repository and maintaining a register of the cloned repository in the database. It does everything asynchronously and, when data is requested, it iterates over every commit of the Master branch of the repository.

Figure 3.12 depicts the architecture made with Sinatra, a library for quickly creating web applications in Ruby. It is responsible for creating the routes to the endpoints and for handling the requests to the application. For Storage, the gem uses an SQLite Database, which is a small, fast, self-contained, free and open-source and easy to use database engine. As it writes its data in a filesystem archive, it is the most used database in the world and has many features that allow the project to be easily maintained.

This micro-service uses the “Rugged”¹⁶ library to make the connection to the Git repositories. It gives access to the libgit2¹⁷ in Ruby. Libgit2 is a pure C implementation of the core methods of Git available through an easy to use and portable API. The current implementation of this application can be found in Appendix B, where it lists all the available endpoints used in this project.

¹⁶<https://github.com/libgit2/rugged>

¹⁷<https://github.com/libgit2/libgit2>

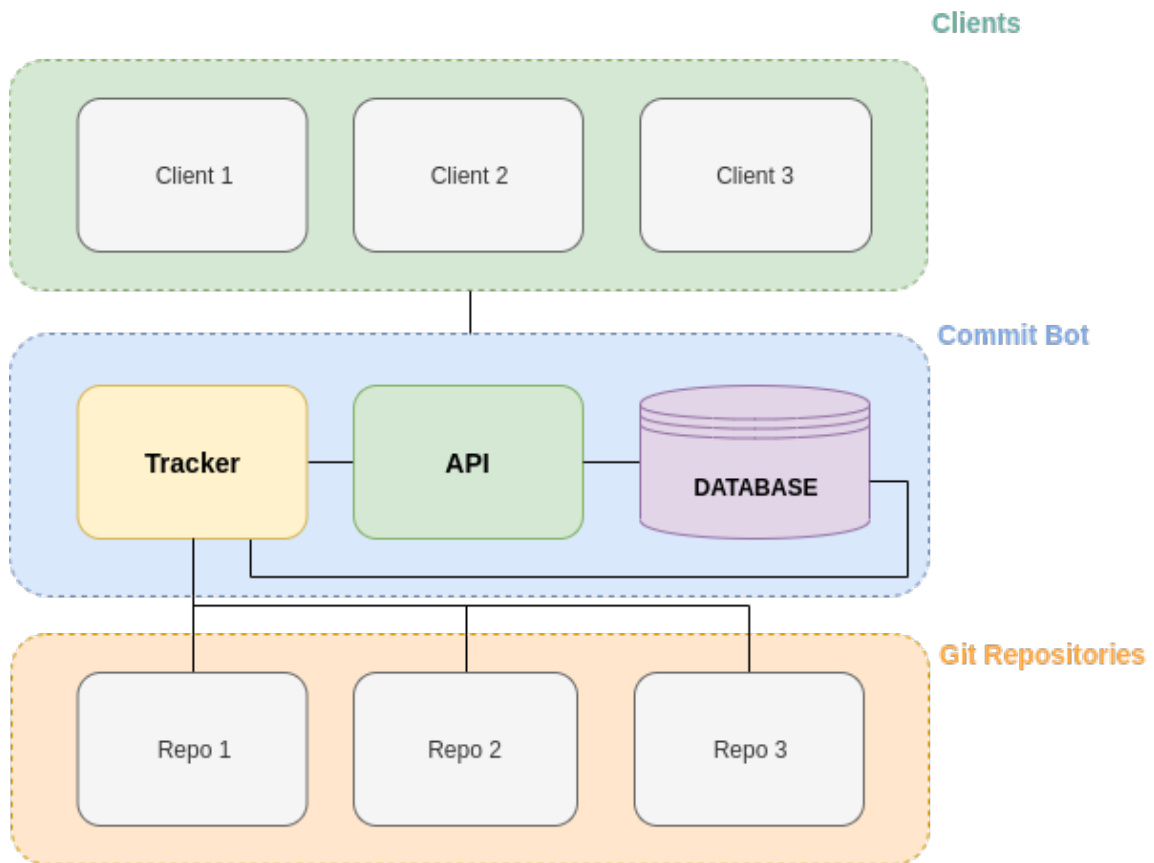


Figure 3.12: *Git bot architecture*

Initially, there was only one endpoint to gather information about a commit, and it returned all of it in a single request. This design was the origin of some problems regarding the performance of the service. When traversing the entire collection of commits, all the data in the commit diffs were being sent – commit diffs are the changes made in a patch. This step wasted a lot of time when the requested information was not a diff. To solve the problem, we divided the big endpoint into smaller ones. Now, we have multiple endpoints for the various kinds of atomic information available and some when the information makes sense to be sent with others. For any further aggregation, the responsibility lies with the integration layer. The next listing shows the big endpoint mentioned earlier:

```

1  {
2    "author": {
3      "email": "andre.tgmello@gmail.com",
4      "name": "Andr\u00e9 Mello"
5    },
6    "committer": {
7      "email": "renatogeh@gmail.com",
8      "name": "Renato Lui Geh"
9    },
10   "date": "2019-07-01T17:48:10-03:00",
11   "diff": "diff --git a/lib/observe_bot.rb b/lib/observe_bot.rb\nindex 7
          c08d8b..9abbbb2 100644\n--- a/lib/observe_bot.rb\n+++ b/lib/
          observe_bot.rb\n@@ -17,9 +17,9 @@ module ObserveChat\n @config.user =
          bot_name\n @config.realname = bot_name\n \n- msg_logger = @db\n+
```

```

        db_instance = @db\n on :channel do |msg|\n- msg_logger.log(msg, server)
        \n+ db_instance.save_message(msg, server)\n end\n end\n \n",
12      "id": "cf1ea74605fc4faecfca4f850d368d45f94dcae6",
13      "message": "Fix message saving function call\n\nThere is no 'log' function
        at the DatabaseWrapper class.\nI believe 'save_message' is what it
        was meant to be here.\n\nSigned-off-by: Andre Mello <andre.
        tgmello@gmail.com>\nSigned-off-by: Renato Lui Geh <renatogeh@gmail.com
        >\n",
14      "subject": "Fix message saving function call"
15    }

```

The fragment above is an example of the original information about a commit.

3.2.7 The IRC Bot

The second micro-service used in the platform was the IRC Bot. Its main responsibility is to register IRC channels and persist their messages in a Database to be available when requested. Like the other services, their principle is the same: it is a ruby application. However, it is not a standalone application as it needs a Data-base to run.

The architecture of the service is simple. First, we have the API Layer taking care of the requests made to the application. Second, the framework used in this bot is the same used to create the Integration layer of the platform, Ruby on Rails. Finally, this layer is responsible for all bots registration and also the orchestration of the tracking bots.

After registering a channel, the service has to maintain a bot always connected to the channel. This happens because of a characteristic of the IRC: when a user becomes inactive for enough time, it is automatically logged out of the channel in most of the clients. To approach this issue, we used the “cinch” framework. Cinch, according to the definition found in the repository, is an “IRC Bot Building Framework for creating IRC bots in Ruby. It provides a simple interface based on plugins and rules. It’s as easy as creating a plugin, defining a rule, and watching your profits flourish”¹⁸. Although the project is a FLOSS, it is no longer maintained by its original authors, meaning that any bugs or defects should be solved by our hands.

For the persistence, we chose Mongo¹⁹, a non-relational, document-oriented, distributed database used for general purposes. Differently from the commit bot, the messages of the IRC channels do not persist for their own; that is, if you, as a user, are logged off from the channel, you lost the messages sent while you were offline. For this reason, we needed a more robust database to use as a historical archive of the IRC channel. The following listing shows the main document of the IRC bot.

For the IRC, we decided to add a Sentiment Analyser feature in each message. We used the gem called “Sentimental”²⁰, which uses a Lexicon Based technique to evaluate. In this gem, the classification is done by unsupervised comparison of a text against Sentiment lexicon that defines the sentiment of each word before their use, for example, lovely words

¹⁸<https://github.com/cinchrb/cinch>

¹⁹<https://www.mongodb.com/>

²⁰<https://rubygems.org/gems/sentimental/versions/1.0.3>

such as “love” and “like” have positive values and their opposites have a negative value. This method is known as Dictionary-based analysis as it needs a collection of words and their synonyms, and the more words collected, the more refined the classification. Even if this method has relatively worse performance than of supervised method (VOHRA and TERAIYA, 2013), the supervised method demands a large amount of labelled training, which would escape from our initial scope of creating the foundation for the project platform. As an example, we have in the following listing classification of ‘1.0’ in the sentiment of the phrase, for our default threshold which is 0, any value above that is positive and below is negative.

```

1  {
2    "_id": {
3      "$oid": "5dafc3eb1471fd268ac631aa"
4    },
5    "nick": "felipe",
6    "message": "I like you",
7    "time": "2019-10-23 03:07:23 UTC",
8    "channel": "#felipecaetanochannel",
9    "server": "irc.freenode.net",
10   "sentiment" 1.0
11  }
```

3.2.8 Email Lists

To gather data about the email lists, we initially used the service in its original form, but some problems were found. For example, to use the service, the user had to use a valid email account to register in a list. This restriction was problematic since it creates overhead to the user of the platform, and it is not in our objective. Another issue in creating this service is the different available archives for mail lists. Nevertheless, making a micro-service to gather all these formats of historical records is another big project with a different scope. Finally, to overcome this problem, the API provides an endpoint in which the user can upload a CSV (comma-separated-values) file to the platform with the following pattern:

```
1  projectId,emailId,senderName,senderEmail,timestampReceived,subject,url,replyto
```

When uploading a file to the Integration Layer, the user can benefit from the endpoints available in the platform for emails.

After all the development process, we ended up with a working interactive website that is backed by a robust back-end. On the next chapter we are going to demonstrate how the website works, by showing it using our chosen example project: Git.

Chapter 4

The Git Community in StarFLOSS

In this chapter, we explain in details how the current StarFLOSS website works, showing how each page works and what they do. The site consists of two different pages, the Home page and the Dashboard page, the following sections are a guide on how to interact with the website.

4.1 The home page

The homepage is a vertical page divided into three sections:

The first section acts as a landing page to the website. This part of the home was already displayed in Section 3.1.2, in Figure 3.2. It is an image banner that is always the size of the users monitor. The only feature it has is the six stars scattered on the picture of the night sky. They are the stars that represent six random projects, they change whenever the user visit the website. If you chose to click the star you are taken to the dashboard of that project, this way you can discover new projects if you feel like adventuring.

The second section is the site explanation section. It explains the reason for the website to exist, the context of FLOSS, and how to read the stars. It explains that the size of a star is the number of members that committed to that project and the size of the pulse, we call it brightness, is the number of commits made.

The third section is a list of all the projects registered to the website (Figure 4.1). It shows a few information on each project and its star. As it is possible to see, we hit a small issue when creating the stars. Because Git has so many members when compared to the other projects, its star becomes the biggest one, and that makes the other smaller projects very small in comparison.

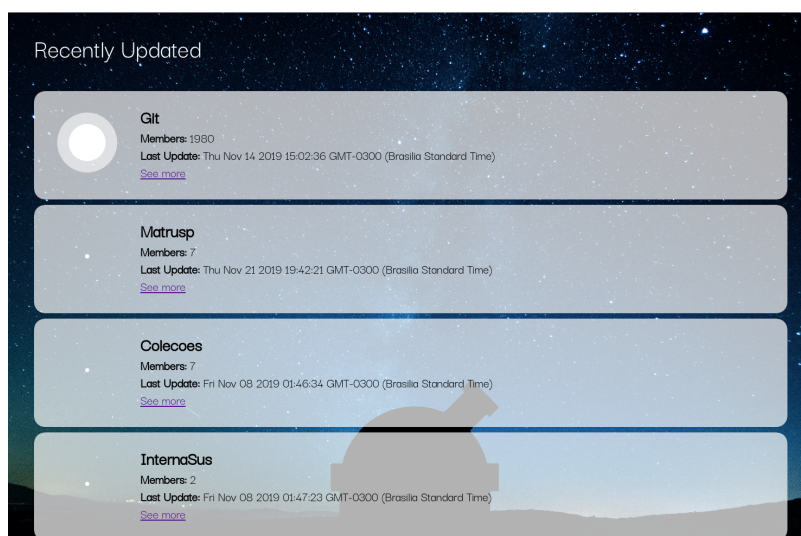


Figure 4.1: List of all projects on the website. In this image it is possible to see the disparity caused by Git.

4.2 The dashboard

The dashboard is self-contained (Figure 4.2). It is divided into two main parts — the side menu, where general information, such as committer name, and the project name, is shared; and the side of the charts, where all the charts are displayed. The chart side is not yet fully responsive as the chart tooltips (extra information that appears on hover) were written in HTML and it does not scale with the SVG drawings of the charts.

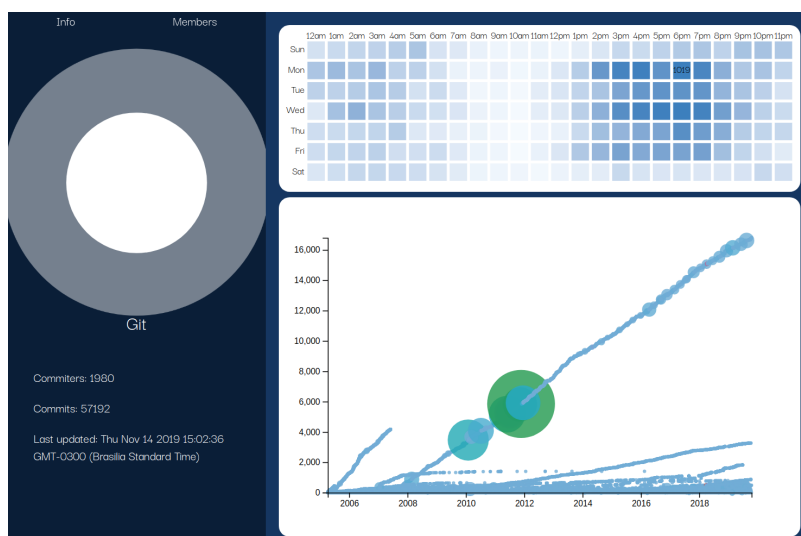


Figure 4.2: StarFLOSS dashboard page for Git.

4.2.1 Information menu

The information menu contains all static info of the selected project (Figure 4.3). The name, number of commits, number of committers and the last time the data has been updated. It is just an overall display for the user to know the project whose information is reflected in the rest of the dashboard.

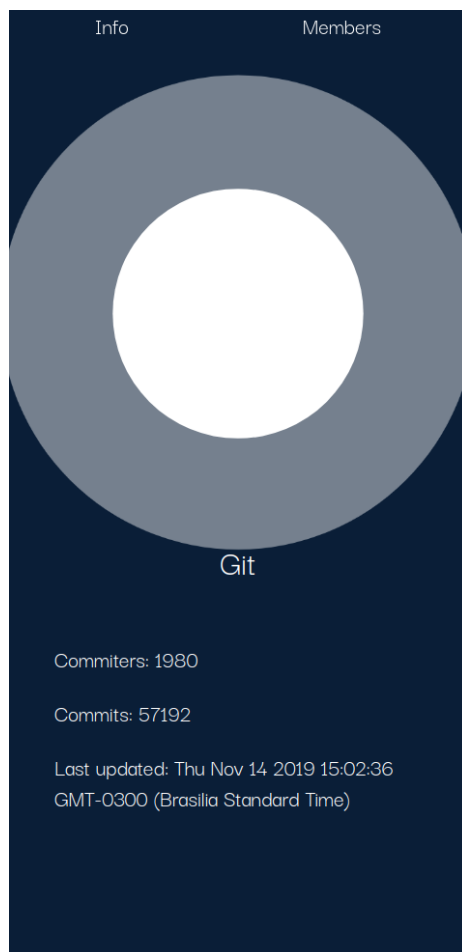


Figure 4.3: *Detail of the Information menu on the dashboard.*

4.2.2 Members menu

The members menu is the second tab of the Information menu. On it is shown a list of all committers to the project in a decrescent order of commits made. Every member listed is clickable if you interact with it both other graphs in the dashboard change to reflect the commits made by only that member (Figure 4.4).

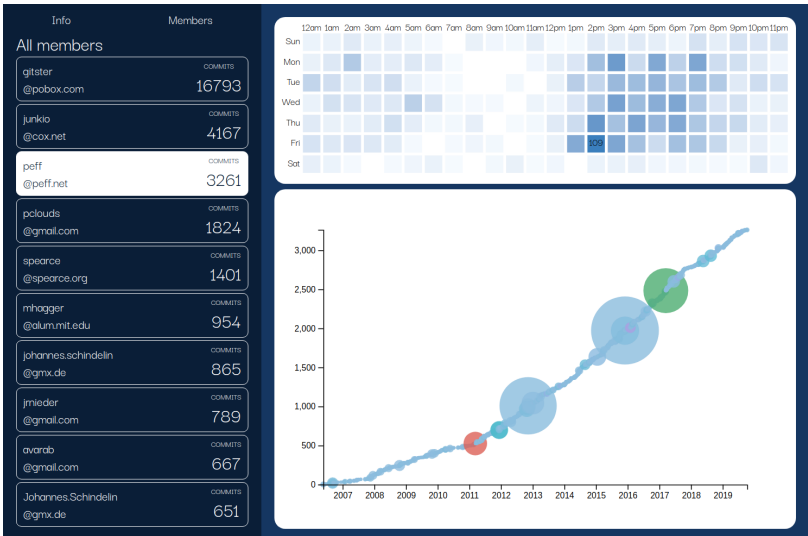


Figure 4.4: *StarFLOSS dashboard. Peff’s was selected so the commits shown on the graphs are only his.*

4.2.3 Time of Day - Heatmap

The heatmap graph shows commits made on a specific weekday and hour (Figure 4.5). This graph was made to see when the community is more active as it shows the period where there were more commits. The hour with most commits is highlighted by having the actual number of commits on display, from there, the colour fades to white, the whiter the colour, the less commits made on that hour.

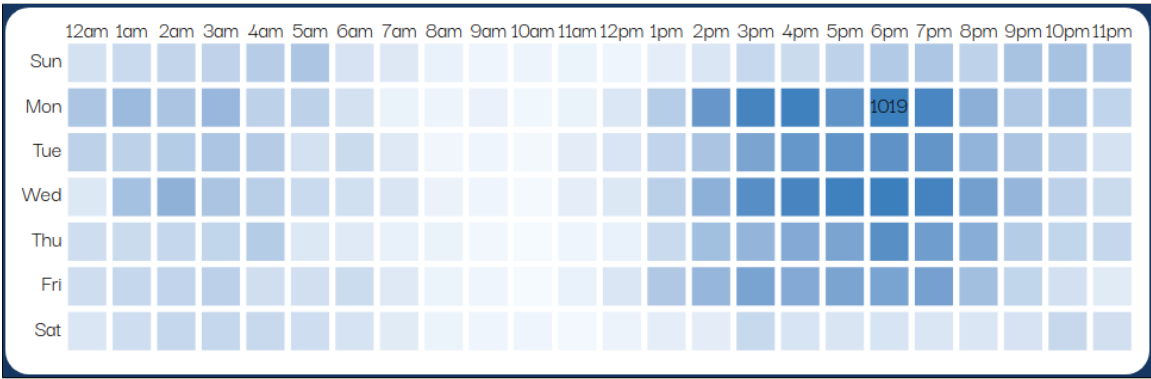


Figure 4.5: *Detail of the Heatmap chart on the dashboard.*

The interaction in this chart comes from clicking the squares. If you hover over them, a tooltip appears showing the top five committers on that hour of that weekday. If you click the chart, the other members menu and bubble chart show only the commits of that hour. If you click more then one square, data from both days will be reflected on the other dashboard parts.

One issue that the heatmap has is the fact that all commits are dated with a 3 GTM. This issue means that they are all transformed into Brasilia’s time-zone. So, they accurately

display the time for Brazil, but not the time it was in the place the committer committed, nor the actual time for other time-zones.

4.2.4 Commits over time - Bubble chart

The bubble chart shows commits per time (Figure 4.6). In the bubble chart, the circles represent commits. The X-axis is a single time axis, and the farther left, the newer they are. The Y-axis represents the total number of commits made by the committer. The lower circles were the first commits made by that committer. The higher one is the latest. The radius of the circle shows how many lines were changed in that commit, the biggest the circle, more lines changed. The colour varies from red to green depending on if there were more additions or deletions on that commit. A commit that only added lines is green, and one that only deleted is red.

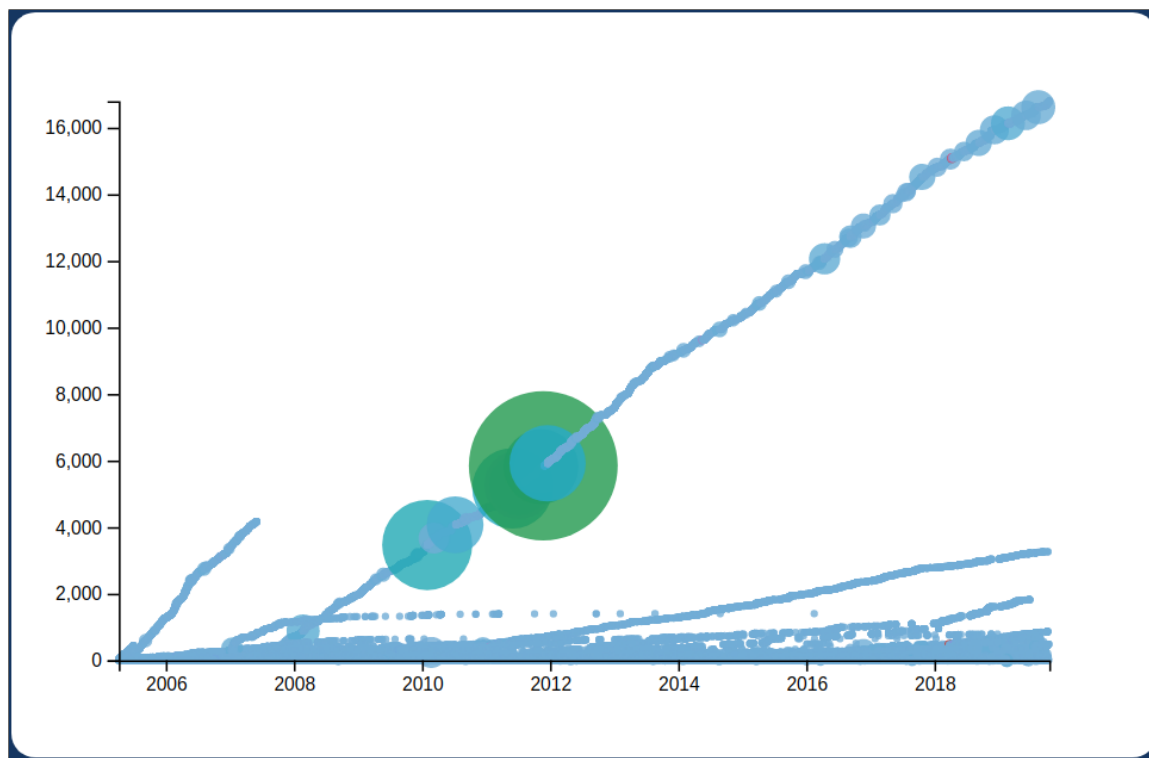


Figure 4.6: Detail of the Bubble chart on the dashboard.

The interaction on this graph, currently, is the same as clicking on a member in the members menu, because it is almost like many line charts, one for each committer. This chart shines when you can see the line of each member. By filtering you can see for example that “gitster” has consistently contributed to Git. While “jrnieder” started committing a lot circa 2010 to 2012 but has been slowing down. On the other hand, “johannes.schindelin” started with a few sparse commits in 2008, but from 2016 up until now has been very consistently committing to Git (see Figure 4.7).

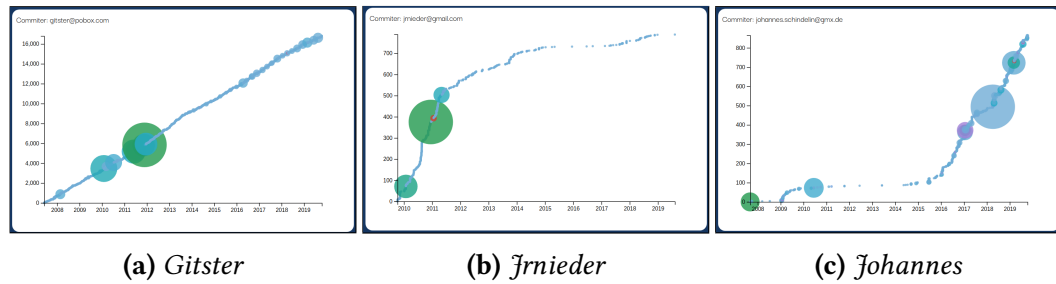


Figure 4.7: Chart for different committers of Git. Notice how the commit pattern changes for each member.

The biggest issue with this chart currently is performance. Every single commit bubble is being rendered on the screen and animated. So, for example, when looking at the Git chart, there are 57,192 different elements drawn and animated. This behaviour causes the site to be hefty for lower performing computers and for the animation to lag whenever it is triggered.

As said in Section 2.2, the Git project was chosen because it was very familiar to us. However, another thing that makes Git a great example project is the long history it has. By using StarFLOSS it is possible to see how years of work transformed Git into what it is today, and to appreciate the thousands of people that participated on this journey.

Chapter 5

Final Remarks

StarFLOSS started looking at a place very different from where it ended. It had to be rebooted after a failed attempt and had a time limit where it needed to stop. Throughout the year, we grew a lot as developers, learning new concepts and tools, and becoming better. Now, we can easily see the mistakes we made in the course of this project. As the famous saying goes “Hindsight is 20/20”, and if we could, there is a lot we would want to do differently.

Regarding the front-end, doing every single component by hand and having a great robust site, in the end, are not complementary ideas, especially in a time crunch. It was a great learning opportunity, and we do not regret taking our time to learn D3. However, StarFLOSS would have more value if the components were taken from Bootstrap or if the charts were made by using eCharts. In the end, function is more than style. Even if having those libraries took some of the charm of the website, it would be more useful if we had done it that way.

Concerning the Integration Layer, the choice for the presented architecture was deemed to be right, since relying only on micro-services for the requisitions would be too much for a starting project. In the context of this work, maintaining the bots was heavy. We worked with different technologies, each one with its learning curves. We felt that the time spent on learning all these technologies was considerable.

Even with those considerations, we view this project as a success. It not only started a development of a tool that can be of much help for further analysis of FLOSS projects but also uses several good practices concerning FLOSS to make contributing easier.

5.1 Where to go from here?

There is no development project that is 100% complete, and the StarFLOSS has a long path to walk before becoming a full-fledged website. The considerations in this last section range from small ideas to huge updates for the whole platform. They are what we would like to do if we had more time to dedicate to this project. We hope the future team that works on continuing this project considers these ideas. However, if new contributors come

with a different mindset of where StarFLOSS could go, we recommend that they follow their vision the same way we followed ours.

5.1.1 The future of the StarFLOSS website

There are two different ways the StarFLOSS website visualisation can go from here. One is to go to a more standard approach for the visualisation, and the other is to go completely crazy on the artistic side.

For a more standardised approach, it is possible to just keep drawing the charts with D3 or change to something easier to draw, such as ChartJs or eCharts. Then create a standard, but useful, dashboard. Aim for something like Kibana or Google Charts. Create good filters and selectors to allow the user to find information quickly, maybe using Flux to facilitate the filtering. Finally, enable the user to download and print essential findings they make. There could also be a dashboard for the whole FLOSS universe, but this is not the main focus of the website. This idea is a common one, but it will be the best design to be used by people who want to get answers. It is aimed to be used by academics who want to find data for studies and FLOSS community members that need to see how their project is advancing.

Otherwise, go completely crazy on the universe's inspiration. It would be fantastic if we could make a galaxy with the FLOSS stars and display it in a 3D environment using the different 3D javascript libraries or follow a more minimalist visual and make it a 2D graph. Have the proximity of each star be a data visualisation in itself, for example, the closer they are, the more members they have in common, and notice whether closer stars are similar projects. Allow the user to click on a star to open the charts on a side menu. And if you want to stretch this idea, have the members be asteroids that navigate between projects, and look for the odd member that travels through super distant stars.

This second idea is more focused on making StarFLOSS a fun website and a display of javascript capabilities. It may not be the best tool to find data, but it will attract users who want to see the vastness of the FLOSS universe, and they would add a new project by a sheer curiosity of where it would land. While the other focuses on the dashboard of a single community, this idea is also good to show how the communities interact with each other.

5.1.2 The future of the StarFLOSS API

Concerning the back-end supporting the platform, we find the architecture of the Integration layer is good the way it is. It uses many micro-services and has the prospect to use many more. It also would be wise to keep this layer with a Gateway to make the impression of having a monolith system.

As in the visualisation, we could have two different strategies to change the usage of the database. First, since many micro-services are to be incorporated into the platform, a conservative approach could be used to facilitate the development of the platform as

a whole. Using a unified database would reduce the number of teams needed to work in the project. It could lead to faster results and ease the trouble of giving maintenance to the project. However, this approach would also lead to some problems as the project would lose the capacity to deploy its services independently, turning the agile practices for continuous integration and deployment considerably tricky.

A second alternative is a more Event-Driven Architecture. It could be implemented offloading the messages to a queue and decoupling the Gateway from the service as it would work as in a Pub-Sub (Publisher, Subscriber) model. The micro-service would be a publisher that would be unaware of the consumer, just keeping on publishing its messages, and the Gateway would be a subscriber to the event stream. This strategy offers us the benefits of still maintaining a micro-services architecture but treating the collection and awareness of new data differently.

Appendix A

Flux pattern

Flux is an application architectural pattern created by Facebook to handle issues with synchronising data through the multiple Models of the Model-View-Controller (MVC) pattern.

The pattern was conceived when the engineer team at Facebook needed to deal with a bug that would make a new message notification be active even if the user had already seen that message. The issue at hand was that the source of truth for the reading event and for the notification did not match, since they were from two different models, so they designed a way to have all data of the website come from a single source, creating the Flux pattern.

The main interest point with Flux is the fact that there is only one source of truth. In Flux, all data pass through an unidirectional flow, when a data change should happen an action is created on the View, this triggers the Dispatcher to change the info on the specific Store, as soon as the Store is changed the View updates. This flow can be seen in Figure A.1.

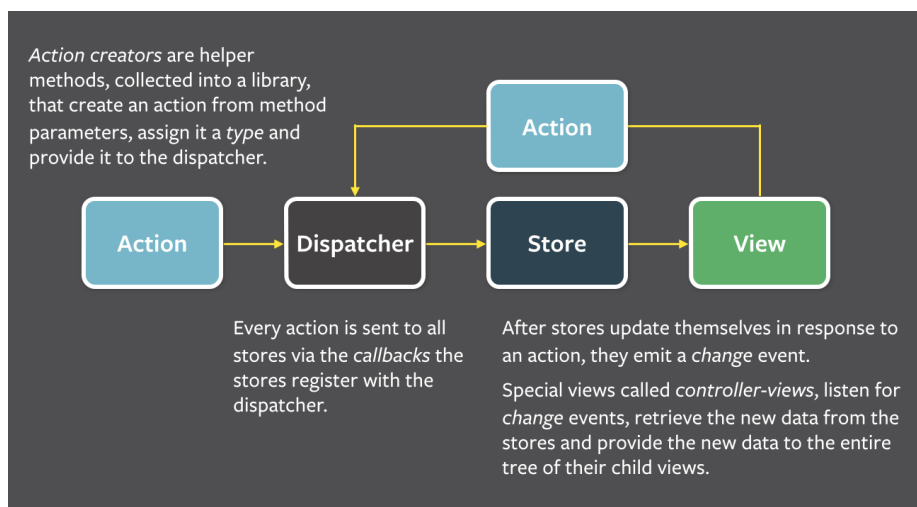


Figure A.1: Flux structure. Source: <https://facebook.github.io/flux/docs/in-depth-overview/>

There are 4 essential parts in the Flux. In the following points we will explain what each one does and show where you can find them being applied on the StarFLOSS front-end¹:

- **Dispatcher:** The dispatcher is one single component that acts as a central hub to all the data changes. It exposes actions and make changes to the Stores. In a program there is one dispatcher, and only they are allowed to change the data on the store.

StarFLOSS dispatcher is located in the Reducers folder (`./src/Store/Reducers`). As we used Redux to implement the pattern they are called Reducers and have a few extra limitations.

- **Actions:** Actions are methods exposed by the dispatcher, by calling an action the dispatcher runs. Through them you may pass the data that needs changing.

StarFLOSS has three implemented actions: `ADD_PROJECT`, `ADD_STATISTIC` and `REMOVE_PROJECT`. The can be found in the ProjectsReducer file (`./src/Store/Reducers/ProjectsReducer.js`)

- **Stores:** A program can have multiple Stores, they act almost as the Model in the MVC pattern. The difference is that a store may hold more than one object, if it makes sense on the page to do so.

StarFLOSS has one Store, it handles the projects data, because of Redux, the store is generated based on the Reducers, so it is not clearly displayed on the code, in the Store file you can find the line that creates it (`./src/Store/Store.js`).

- **Views:** The View is the code that generate the screen that the user interacts with. Using React the View does the paper of Controller as well in an MVC. The view calls actions depending on the user input.

As a React project StarFLOSS is made of Views. However to better exemplify the use of Flux and Redux, we suggest taking a look the Home code file (`./src/Components/Home/Home.js`), where in line 34 we add new projects to the store and the Ranking code file (`./src/Components/Home/Ranking/Ranking.js`) where in line 36 we read from the store. You can see that in the end of both of the files there are some bindings that had to be added, that is to link the View to the Store and to the Dispatcher/Reducer.

¹All code for StarFLOSS front-end can be found on the following Gitlab repository: <https://gitlab.com/flusp/msfloss/msfloss-interface>

Appendix B

Endpoints

B.1 Commit Information Endpoints

- **GET /api/v1/projects/:project_id/commits**

Get an array with all the commits of the project

Response body example:

```

1  {
2    "id": "cf1ea74605fc4faecfca4f850d368d45f94dcae6",
3    "date": "2019-07-01T20:48:10+00:00",
4    "author_email": "felipe.caetano@gmail.com",
5    "committer_name": "Felipe Caetano",
6    "subject": "Fix message saving function call"
7  }
```

- **POST /api/v1/projects/:project_id/mycommits**

Get an array with all the commits of the author email

Request body example:

```

1  {
2    "email": {
3      "author_email": "felipe.caetano@gmail.com"
4    }
5  }
```

Response body example:

```

1  {
2    "id": "cf1ea74605fc4faecfca4f850d368d45f94dcae6",
3    "date": "2019-07-01T20:48:10+00:00",
4    "author_email": "felipe.caetano@gmail.com",
5    "committer_name": "Felipe Caetano",
6    "subject": "Fix message saving function call"
7  }
```

- **GET /api/v1/projects/:project_id/commits/statistics**

Get total number of commits, distinct committers and number of commits per user

Response body example:

```

1  {
2    "total_commits": 134,
3    "distinct_members": 11,
4    "total_by_member": {
5      "andre.bulha@gmail.com": 3,
6      "bianca@gmail.com": 38,
7      "caraf@gmail.com": 2,
8      "camel.a.anjos@gmail.com": 24,
9      "junit.scaroni@gmail.com": 72,
10     "carossela.dm3@gmail.com": 13,
11     "brusque@MacBook-Sergio.local": 1,
12     "cartao@gmail.com": 1,
13     "ostra@gmail.com": 4
14   }

```

- **GET /api/v1/projects/:project_id/commits/diffs**

Get an array representing users and its own commits, with information about the dates and number of altered lines

Response body example:

```

1  {
2    "name": "felipe.caetano@gmail.com",
3    "commits": [
4      {
5        "date": "2019-07-01T20:48:10+00:00",
6        "author_name": "felipe.caetano@gmail.com",
7        "added_lines": 2,
8        "deleted_lines": 2
9      }
10   ],
11 },
12 {
13   "name": "camila.naomi@gmail.com",
14   "commits": [
15     {
16       "date": "2019-07-01T20:48:10+00:00",
17       "author_name": "camila.naomi@gmail.com",
18       "added_lines": 2,
19       "deleted_lines": 2
20     }
21   ]

```

B.2 Email Information Endpoints

- **GET /api/v1/projects/:project_id/emails**

Get an array with all the emails of the uploaded emailList

Response body example:

```

1  [
2  {
3    "projectId": 1,
4    "emailId": "12341",
5    "senderEmail": "felipe.caetano@usp.br",
6    "senderName": "Felipe Caetano",
7    "timestampReceived": "2019-07-01T20:48:10+00:00",
8    "subject": "Questions about a patch",
9    "url": "http://marc.info/?l=git&m=156995454523991&w=2",
10   "replyto": "http://marc.info/?l=git&m=156995454523991&w=2"
11  }
12  ]

```

- **POST /api/v1/projects/:project_id/myemails**

Get an array with all the emails from specific sender

Request body example:

```

1  {
2    "email": {
3      "author_email": "naomi.camila@usp.br"
4    }
5  }

```

Response body example:

```

1  {
2    "projectId": 1,
3    "emailId": "12341",
4    "senderEmail": "felipe.caetano@usp.br",
5    "senderName": "Felipe Caetano",
6    "timestampReceived": "2019-07-01T20:48:10+00:00",
7    "subject": "Questions about a patch",
8    "url": "http://marc.info/?l=git&m=156995454523991&w=2",
9    "replyto": "http://marc.info/?l=git&m=156995454523991&w=2"
10  }

```

- **GET /api/v1/projects/:project_id/emails/statistics**

Get total number of emails, distinct senders and number of emails sent per user

Response body example:

```

1  {
2    "total_emails": 134,
3    "distinct_senders": 11,
4    "total_by_sender": {
5      "andre.bulha@gmail.com": 3,
6      "bianca@gmail.com": 38,
7      "caraf@gmail.com": 2,
8      "camel.a.anjos@gmail.com": 24,
9      "junit.scaroni@gmail.com": 72,
10     "carossela.dm3@gmail.com": 13,

```

```

11         "brusque@MacBook-Sergio.local": 1,
12         "cartao@gmail.com": 1,
13         "ostra@gmail.com": 4
14     }
15 }

```

B.3 IRC Information Endpoints

- **GET /api/v1/projects/:project_id/ircs**

Get an array with all the irc messages from the irc channel of a project

Response body example:

```

1
2  {
3      "_id": {
4          "$oid": "5dafc3eb1471fd268ac631aa"
5      },
6      "nick": "júlio Alves",
7      "message": "I wanna try something new",
8      "Time": "2019-07-01T20:48:10+00:00",
9      "Server": "irc.freenode.net",
10     "Channel": "new-channel"
11 }

```

- **POST /api/v1/projects/:project_id/ircs/mymessages**

Get an array with all the messages from specific nick

Request body example:

```

1  {
2      "nick": {
3          "name": "felipec"
4      }
5  }

```

Response body example:

```

1  {
2      "_id": {
3          "$oid": "5dafc3eb1471fd268ac631aa"
4      },
5      "nick": "felipec",
6      "message": "I wanna try something new",
7      "Time": "2019-07-01T20:48:10+00:00",
8      "Server": "irc.freenode.net",
9      "Channel": "new-channel"
10 }

```

- **GET /api/v1/projects/:project_id/ircs/statistics**

Get total number of messages sent, distinct senders and number of messages sent per user

Response body example:

```
1  {
2    "total_messages": 43,
3    "distinct_nicks": 3,
4    "total_by_nick": {
5      "felipe": 3,
6      "juliaj": 38,
7      "caramelo": 2
8    }
```


References

- [BARCOMB et al. 2019] Ann BARCOMB, Klaas-Jan STOL, Dirk RIEHLE, and Brian FITZGERALD. “Why Do Episodic Volunteers Stay in FLOSS Communities?” In: ICSE ’19. Montreal, Quebec, Canada, 2019, pp. 948–954 (cit. on p. 1).
- [BILL SHANDER 2016] BILL SHANDER. *Data Visualization: Storytelling*. chapter 3: Story Mechanisms, section 6: Personalization. Aug. 2016. URL: <https://www.linkedin.com/learning/data-visualization-storytelling/personalization-2> (visited on 10/30/2019) (cit. on p. 10).
- [BRAD FROST 2013] BRAD FROST. *Development is Design*. Oct. 2013. URL: <https://bradfrost.com/blog/post/development-is-design/> (visited on 11/22/2019) (cit. on p. 15).
- [CROWSTON, WEI, et al. 2012] Kevin CROWSTON, Kangning WEI, James HOWISON, and Andrea WIGGINS. “Free/Libre open-source software development”. In: *ACM Computing Surveys* 44.2 (Feb. 2012), pp. 1–35 (cit. on p. 1).
- [CROWSTON and SQUIRE 2017] Kevin CROWSTON and Megan SQUIRE. “Lessons Learned from a Decade of FLOSS Data Collection”. In: *Big Data Factories: Collaborative Approaches*. Ed. by Sorin Adam MATEI, Nicolas JULLIEN, and Sean P GOGGINS. Cham: Springer International Publishing, 2017, pp. 79–100 (cit. on p. 1).
- [DRAGONI 2016] Nicola DRAGONI. “Microservices: yesterday today and tomorrow”. In: (2016) (cit. on p. 24).
- [DUCHENEAUT 2006] Nicolas DUCHENEAUT. “Socialization in an Open Source Software Community: A Socio-Technical Analysis”. In: (2006) (cit. on pp. 8, 9).
- [ELIJAH MEEKS 2018] ELIJAH MEEKS. *D3 is not a Data Visualization Library*. June 2018. URL: https://medium.com/@Elijah_Meeks/d3-is-not-a-data-visualization-library-67ba549e8520 (visited on 11/25/2019) (cit. on p. 20).
- [FITZGERALD 2006] Brian FITZGERALD. “The Transformation of Open Source Software”. In: (2006) (cit. on p. 1).
- [FSF 2007] FREE SOFTWARE FOUNDATION. *What is free software?* 2007. URL: www.gnu.org/philosophy/free-sw.en.html (visited on 10/30/2019) (cit. on p. 5).

- [INA SALTZ 2013] INA SALTZ. *Graphic Design Foundations: Typography*. chapter 1: Introduction, section 2: Why good typography matters. Feb. 2013. URL: <https://www.linkedin.com/learning/graphic-design-foundations-typography/why-good-typography-matters> (visited on 11/20/2019) (cit. on p. 16).
- [JESUS M. GONZALEZ-BARAHONA and COSENTINO 2018] Gregorio Robles JESUS M. GONZALEZ-BARAHONA Santiago Dueñas and Valerio COSENTINO. “Perceval: Software Project Data at Your Will”. In: (2018) (cit. on p. 8).
- [KON et al. 2011] Fabio KON et al. “Free and Open Source Software Development and Research: Opportunities for Software Engineering.” In: *SBES*. IEEE Computer Society, 2011, pp. 82–91. ISBN: 978-1-4577-2187-8 (cit. on p. 1).
- [MICHAEL FRIENDLY 2006] MICHAEL FRIENDLY. “A Brief History of Data Visualization”. In: *Handbook of Computational Statistics: Data Visualization*. Ed. by C. CHEN, W. HÄRDLE, and A UNWIN. Vol. III. Heidelberg: Springer-Verlag, 2006 (cit. on p. 15).
- [ØSTERLIE and JACCHERI 2007] Thomas ØSTERLIE and Letizia JACCHERI. “A Critical Review of Software Engineering Research on Open Source Software Development”. In: (2007) (cit. on p. 1).
- [RAYMOND 1997] Eric Steven RAYMOND. “The Cathedral and the Bazaar”. In: (1997) (cit. on p. 1).
- [SCACCHI 2010] Walt SCACCHI. “The Future of Research in Free/Open Source Software Development”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER ’10. New York, NY, USA: ACM, 2010, pp. 315–320 (cit. on p. 1).
- [STEINMACHER et al. 2015] Igor STEINMACHER, Marco Aurelio GRACIOTTO SILVA, Marco Aurelio GEROSA, and David F. REDMILES. “A systematic literature review on the barriers faced by newcomers to open source software projects”. In: *Information and Software Technology* 59 (Mar. 2015), pp. 67–85 (cit. on p. 1).
- [TONY HARMER 2018] TONY HARMER. *Introduction to Graphic Design*. Feb. 2018. URL: <https://www.linkedin.com/learning/introduction-to-graphic-design-3/> (visited on 11/20/2019) (cit. on p. 15).
- [VOHRA and TERAIIYA 2013] S. M. VOHRA and J. B. TERAIIYA. “A COMPARATIVE STUDY OF SENTIMENT ANALYSIS TECHNIQUES”. In: (2013) (cit. on p. 32).
- [YEFIM NATIS 1996] Roy Schulte e YEFIM NATIS. “Service Oriented Architectures”. In: (1996) (cit. on p. 24).