PUC

ISSN 0103-9741

Monografias em Ciência da Computação

nº 01/20

GHFramework - A Framework to Collect and Manipulate GitHub's API Data

Caio Barbosa

Alessandro Garcia (advisor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 01/20 ISSN: 0103-9741
Prof. Marcos Kalinowski Julho,2020

GHFramework - A Framework to Collect and Manipulate GitHub's API Data

Caio Barbosa Alessandro Garcia (advisor)

csilva@inf.puc-rio.br ,

Resumo. A programação colaborativa está presente em todo o desenvolvimento de software moderno e, quando você pensa sobre isso, a plataforma GitHub é um dos primeiros nomes que vem à mente. Muitos estudos no campo da engenharia de software estão focados na análise de repositórios do GitHub. Além disso, a plataforma fornece APIs para automatizar algumas ações (por exemplo, enviar e recuperar dados), que são usadas nesses estudos para coleta de dados. Para automatizar esse processo de coleta, foram construídas algumas ferramentas (RepoDriller e PyDriller) e um banco de dados (GHTorrent). Contudo, existem limitações no RepoDriller e PyDriller, o desenvolvedor tem acesso apenas aos dados do Git, impossibilitando o acesso à informações de Issue e Pull Requests; o GHTorrent é apenas um banco de dados, e suas consultas são feitas utilizando códigos em Pearl. Este trabalho propõe a criação de um arcabouço de código para auxiliar engenheiros e pesquisadores de software a coletar e manipular os dados da API do GitHub sem seu conhecimento adequado. Além disso, preparando o caminho para eles se concentrarem em suas análises e não gastarem recursos em consultas de API.

Palavras-chave: Mineração de Dados, Repositório de Código, API, GitHub, Arcabouço de Software

Abstract. Collaborative programming is everywhere in modern software development, and, when you think about it, the GitHub platform is one of the first names that comes to mind. Many studies in the field of software engineering are focused on the analysis of GitHub repositories. Moreover, the GitHub platform provides APIs to automate some actions (e.g. sending and retrieving data), which is used on those studies for data collection. In order to automate this process of data collection, some tools (RepoDriller and PyDriller) and a dataset (GHTorrent) were built. Although, there are limitations on RepoDriller and PyDriller the developer has only access to data from Git; the GHTorrent is only a dataset, that can only be queried using Pearl. This work proposes the creation of a framework to assist software engineers and researchers to collect and manipulate the data of the GitHub API without its proper knowledge. Also, paving the way for them to focus on their analysis and not spending resources on API querying.

Keywords: Data Mining, Code Repository, API, GitHub, Framework

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Contents

1 Introduction

Collaborative programming is everywhere in modern software development, people work from their homes, their own countries, and are not required being in the office to work in a company or together with a team. In this world, the GitHub [1] platform is the first name that comes to mind: "git commit, git push" is practically a law, and the numbers speak for themselves: more than 100+ million projects, 31+ million users and 2+ million organizations are hosted in it [1]. Many studies in the field of software engineering [2–5] are focused on the analysis of GitHub repositories. These studies analyze the commits, issues, pull requests, discussions (commentaries), developers, and, to do so, they have to collect data from the GitHub. Furthermore, they have to use some programming language to bind objects, manipulate the data and create metrics [6], to quantify this data, build prediction models [7], and analyze some information, *e.g.*, the developer's best practices [8].

The GitHub platform provides a REST [2] (v3) and a GraphQL [3] (v4) APIs to automate some actions (*e.g.*, sending and retrieving data). However, these API are complex and the documentation lacks more precise information about the objects being accessed. The GitHub APIs (v3) lack of data manipulation, and also, only deals with sending and retrieving data in the JSON format. Yet, the GitHub (v4) API was made to better handle with this data manipulation, since, instead of collecting all the data and the whole objects, it lets you query which data you want. However, it also needs a programming language working below it for data binding and the developer knowledge and understanding of the API.

This work proposes the creation of a framework to assist software engineers and researchers to collect and manipulate the data of the GitHub API without its proper knowledge. Also, paving the way to focus on the analysis and not spending resources on API querying.

The structure of the remainder of this document is as follows. In **Section 2**, the GitHub API will be better introduced; **Section 3** brings the details about the proposed framework; **Section 4** presents the framework diagrams; **Section 5** shows the tests cases processed for the framework;**Section 6** shows how to proper run it; and, finally, **Section 7** concludes.

[1] https://github.com
[2] https://developer.github.com/v3/
[3] https://developer.github.com/v4/

2 Background and Related Work

2.1 GitHub API

Before rushing into the details of the framework (Section 3), there is a need of showing how the GitHub API works, which kind of data is available and give some insight about the endpoints of the API. Currently, there are are two versions of the GitHub API available to use: the REST API (v3) and the GraphQL API (v4). In the REST API all data is sent and retrieved as JSON files and the base URL for the API is "https://api.github.com". The GraphQL API, that is still in early access, is based on a built-in query language, supporting the user to build a query to fetch exactly the data desired, instead of requests to an endpoint. However, even with this data-focused API, there's still a need for some language to make the queries and manipulate the data. Since this work was done using the v3, the information further discussed will be only related to this version of the API.

Looking into the resources available, there are plenty of endpoints, for various types of information, from commits and issues to events and milestones. With the intention of clarifying the data used in this work, only the endpoints used will be addressed.

2.1.1 Commits

On the endpoint Commits [4] just two actions are being used: (I) List commits on a repository and (II) Get a single commit. On (I), given a repository, this endpoint returns a JSON file with a list with the last 30 commits of the repository and, by adding pagination to the endpoint, it will return a list with the next 30 commits until the first commit of the repository. On (II), given a commit SHA (unique ID), this endpoint returns a JSON file with the information of the commit.

2.1.2 Comments

The Comments [5] endpoint has two actions being used on this work: (I) List comments on a issue and (II) Get a single comment. On (I), given an issue number, this endpoint returns a JSON file with a list with the first 30 comments of the issue and, by adding pagination to the endpoint, it will return a list with the next 30 comments until the last comment of the issue. On (II), given a comment id and a repository, this endpoint returns a JSON file with the information of the comment.

2.1.3 Issues

Endpoint Issues [6], just two actions are being used: (I) List issues for a repository and (II) Get a single issue. On (I), given a repository, this endpoint returns a JSON file with a list with the last 30 issues of the repository and, by adding pagination to the endpoint, it will return a list with the next 30 issues until the first issue of the repository. On (II), given an issue number and a repository, this endpoint returns a JSON file with the information of the issue.

[4] https://developer.github.com/v3/repos/commits/
[5] https://developer.github.com/v3/issues/comments/
[6] https://developer.github.com/v3/issues/

2.1.4 Pull Requests

From the Pull Requests [7]: (I) List pull requests and (II) Get a single pull request. On (I), given a repository, this endpoint returns a JSON file with a list with the last 30 pull requests of the repository and, by adding pagination to the endpoint, it will return a list with the next 30 pull requests until the first pull request of the repository. On (II), given a pull request number and a repository, this endpoint returns a JSON file with the information of the pull request.

2.1.5 Issue Events

Finally. on the endpoint Issue Events [8]: collect events triggered by activities in issues and pull requests.

2.2 Related Work

Many studies in the field of software engineering [2–5] are focused on the analysis of GitHub repositories. These studies analyze the commits, issues, pull requests, discussions (commentaries), developers, and, to do so, they have to collect data from the GitHub. Furthermore, they have to use some programming language to bind objects, manipulate the data and create metrics [6], to quantify this data, build prediction models [7], and analyze some information, *e.g.*, the developer's best practices [8]. Previous works [9, 10] do an outstanding job concerning repository mining, but, they are limited to data from Git. Also, there is [11], an enormous dataset created from the GitHub API, that is very complicated to manipulate since its queries need to be done on the Pearl language.

[7] https://developer.github.com/v3/pulls/
[8] https://developer.github.com/v3/issues/events/

3 GHFramework

3.1 Framework Purpose

Previous works (Section 2.2 many times have to write their own scripts to collect this data, this make them to spend time in something that should be already done for them. Their focus should be on the data analysis, not the data collection.

The GHFramework was built to present to this researchers the solution for data collection on the GitHub API. The actual implementation of GHFramework is capable of collect data from the following endpoints:

- Commits
- Issues
- Pull Requests
- Comments
- Events

Moreover, the GHFramework also have a prototype class for the endpoint data collection. So the user can easily reach other endpoints without major effort.

More information about each endpoint can be seen on Section 2.

3.2 Use Cases

Section 2 address many works that use the GitHub API to collect data. The use of GHFramework benefit these researchers, making their workflow easier. Taking the work of [8] in consideration, the amount of data collected from GitHub was huge, using the GHFramework to collect it could have saved them a lot of time. Moreover, the GHTorrent [11] can also be improved if their maintainers use the GHFramework to automate the data collection.

3.3 GitHub Metrics

In order to help new researchers using this framework, there is an initial set of 11 metrics implemented. The list and description of each metric can be seen on Table 1.

Table 1: Metrics available on the GHFramework.

Metrics	Description
Number of Users	Number of unique users that interacted in any way in a discussion inside an Issue or Pull Requests (opened, commented, merged or closed)
Number of Contributors	Number of unique contributors that interacted in any way in an Issue or Pull Request (opened, commented, merged or closed)
Number of Core Developers	Number of unique core developers that interacted in any way in an Issue or Pull Request (opened, commented, merged or closed)
Pull Request Opened By	The type of user that has opened each issue or pull request. The user might be an Employee or Temporary. Employees are active contributors and code developers. Conversely, temporary are contributors that do not actively work on the project or does not work for the software organization.
Number of Comments	Number of comments inside an Issue or Pull Request.
Mean Time Between Comments	Sum of the time between all comments of an Issue or Pull Request weighted by the number of comments.
Discussion Length	Time in days that an Issue or Pull Request lasted (difference of creation and closing days).
Number of Snippets in Discussion	The number of snippets inside each comment of an Issue or Pull Requests. Those snippets are detected by the number of ''' (syntax that opens a snippet in markdown) divided by two (opening and closing).
Snippet Size	Sum of the size of all snippets found on comments in an Issue or Pull Request.
Number of Words in Discussion	Sum of the all words of each comment inside an Issue or Pull Request. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.
Number of Words per Comment in Discussion	Sum of the all words of each comment inside a Pull Request weighted by the number of comments. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.

4 Modelling GHFramework

This section presents the modelling of GHFramework based on UML diagrams. Section 4.2 presents the use case diagram of the software framework. Section 4.1 presents the class diagram. Finally, Section 4.3 presents a sequence diagram that illustrates how the software is used.

4.1 Class Diagram

The framework contains four main packages: (i) API; (ii) Utils; and (iii) Metrics. The API contains the classes used to collect data from each endpoint and their respective DAO (Data Access Objects) used to gather selected information on the collected JSONs; Utils contains the classes responsible for connecting the API and handling the I/O of the JSONs and CSVs; finally, the Metrics package contains the implementation of the initial pack of metrics to be used on the framework. We can see the Class Diagram of the system on Figures [1-4]

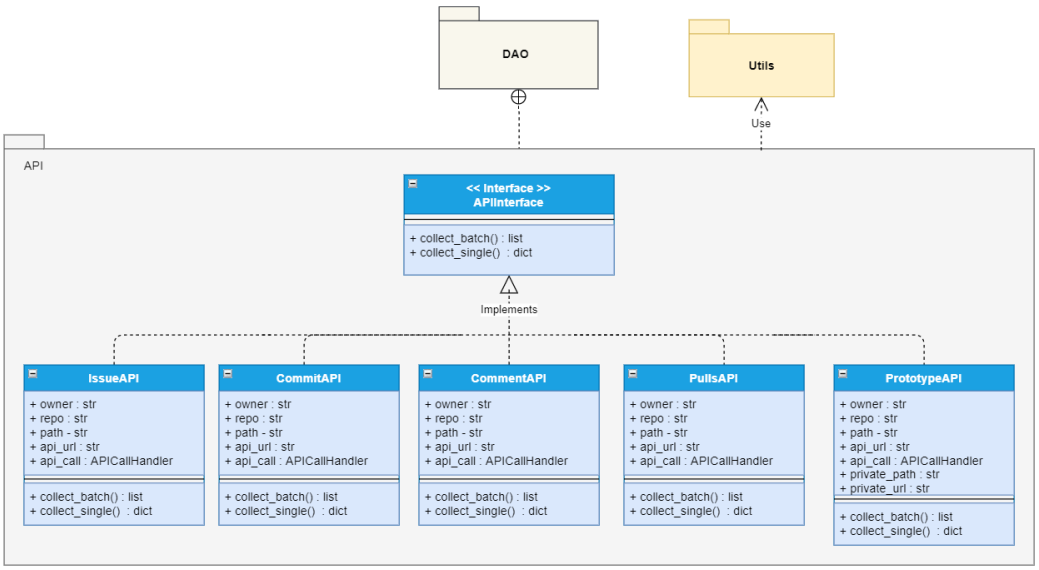

Figure 1: Class diagram of the GHFramework

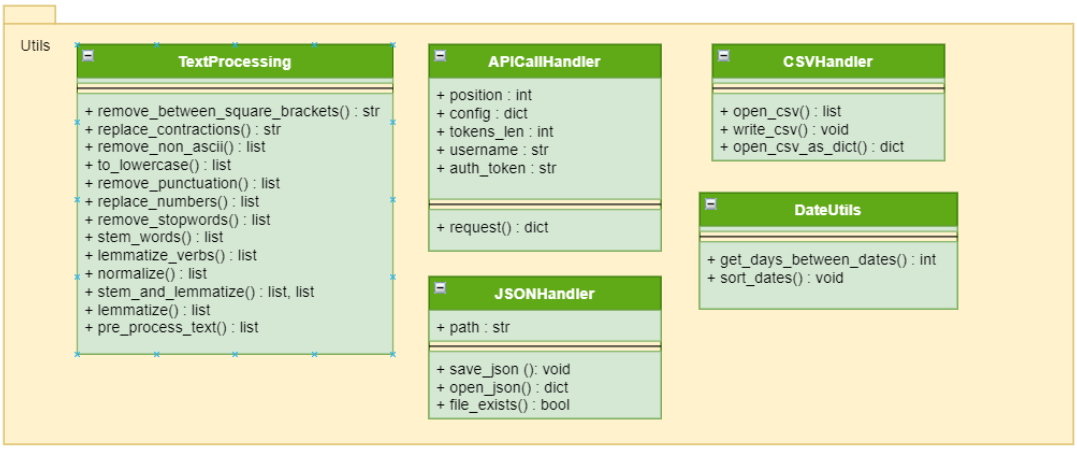

Figure 2: Class diagram of the GHFramework

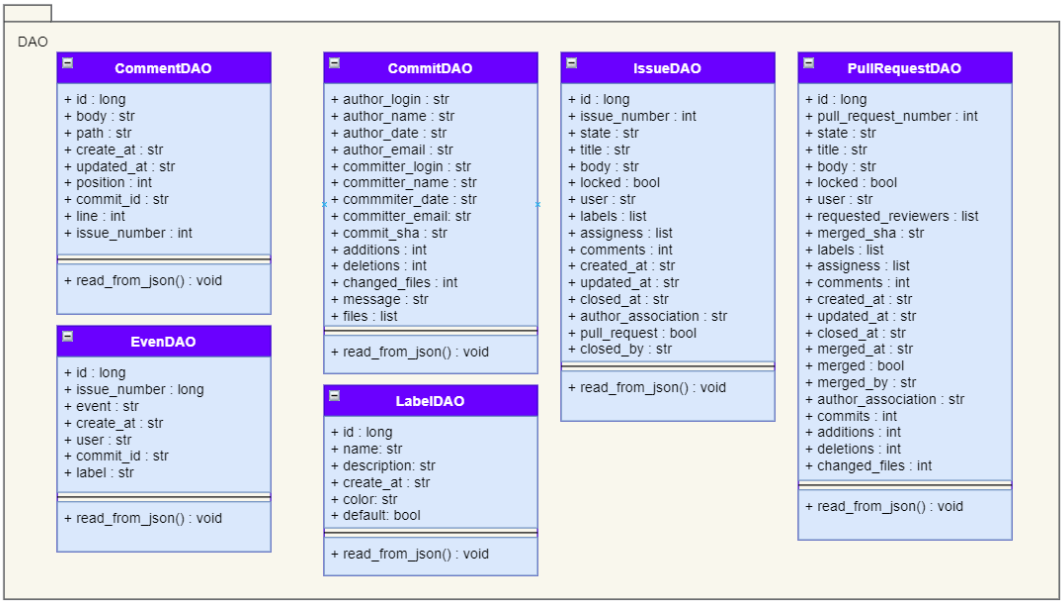

Figure 3: Class diagram of the GHFramework

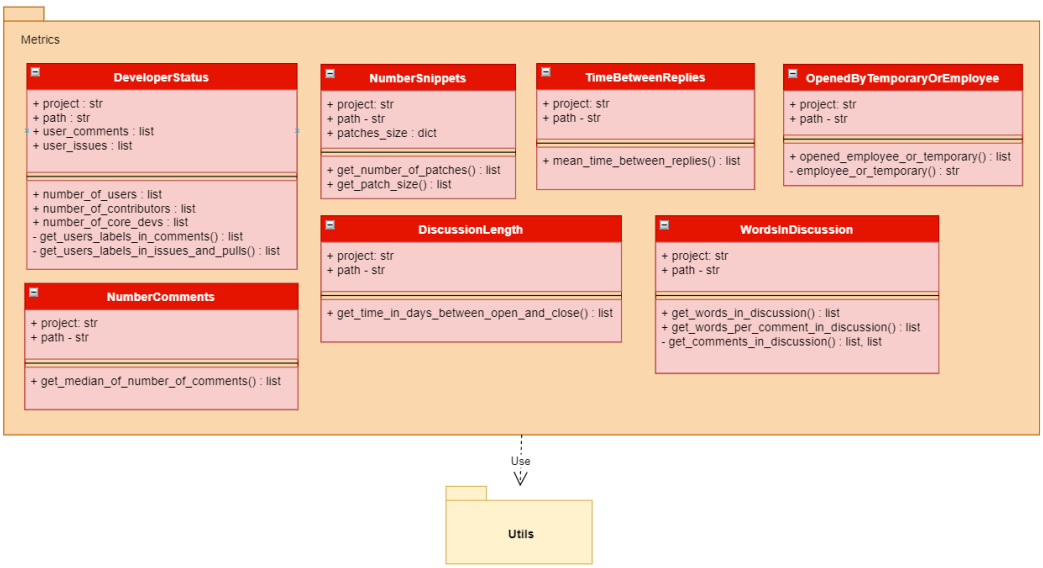

Figure 4: Class diagram of the GHFramework

4.2 Use Case Diagram

Figure 5 presents the software framework's general use case diagram. This diagram has three packages, namely **API**, **Utils** and **Metrics**. The first package has five actions that and a software developer(the main actor) might perform: **Collect Issues**, **Collect Pull Requests**, **Collect Events**, and **Collect Comments** and **Collect Commits**. The second package represents examples of the actions that occur in background by the user action. The API actions are hot spots, the users can implement their only call to the API, as the Utils, is part Frozen Spot and part Hot Spot, the API Request is Hot and the persistence part (Open/Save JSON) is also a Frozen Spot. The metrics are a hot spot, since new metrics can be added to the framework.

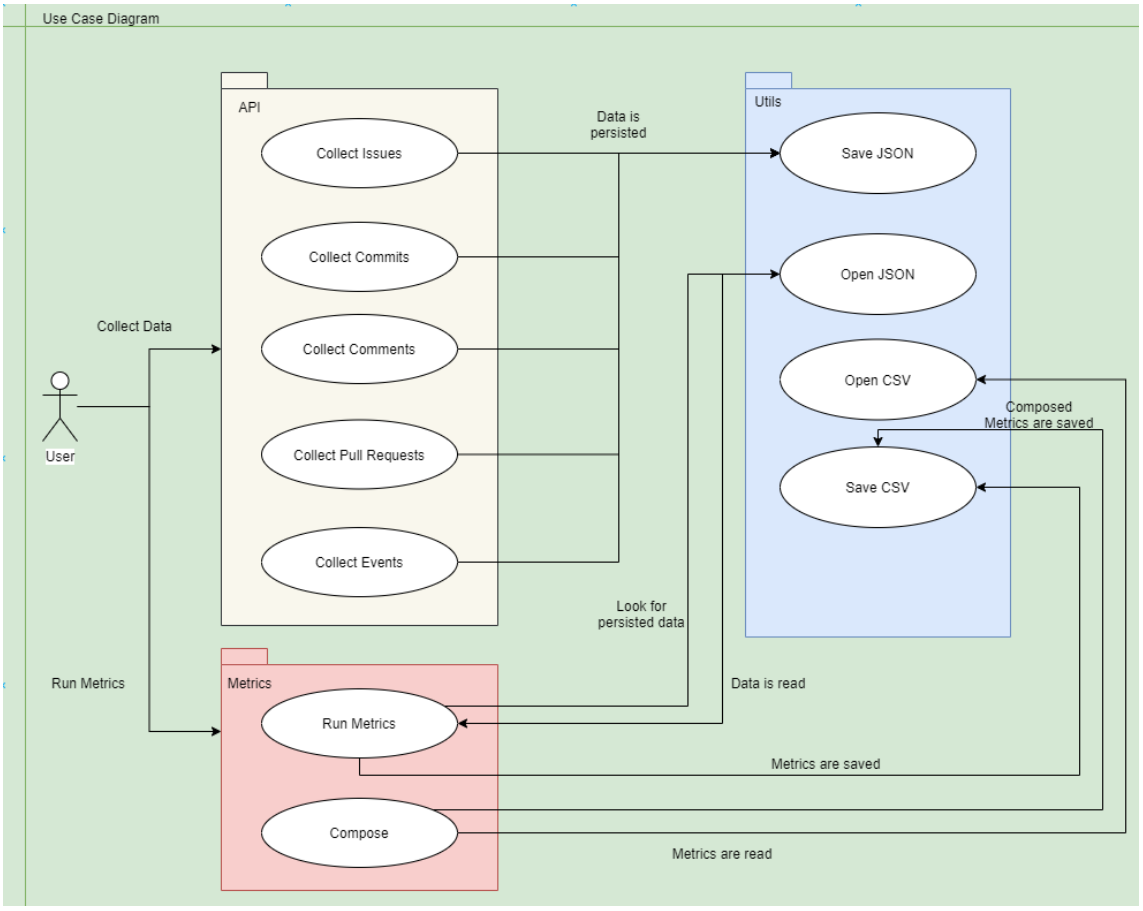

Figure 5: Use case diagram of GHFramework

4.3 Sequence Diagram

Figure 6 presents the sequence diagram of our software framework, specifically for collecting the data from the GitHub API and the persistence involved.

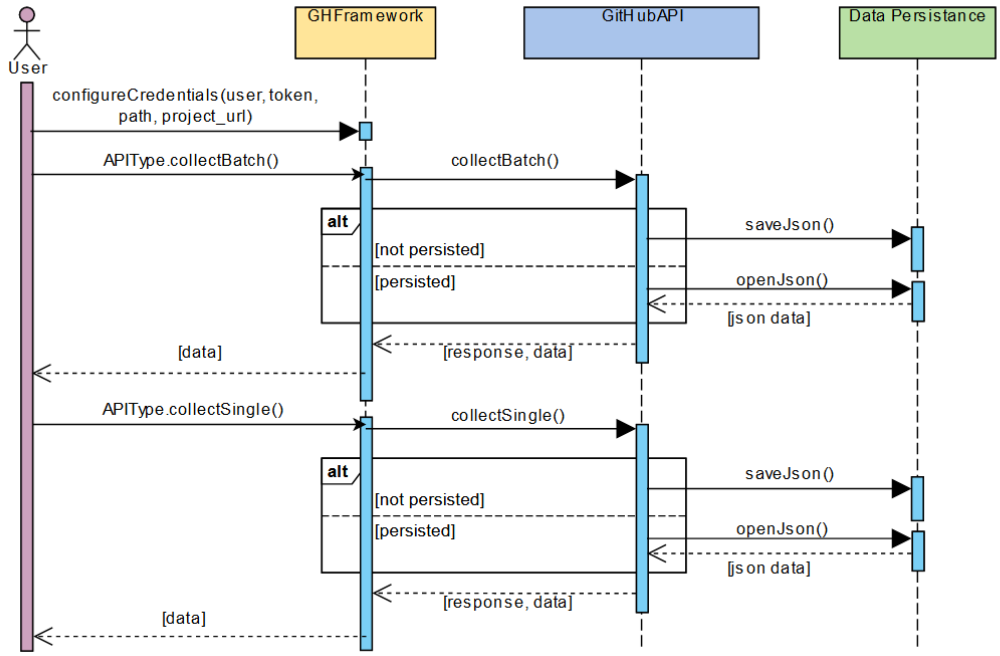

Figure 6: Sequence diagram of our software framework

5 Test Cases

The test cases implemented for the GHFramework consisted in two main classes: APICollector and MetricsCollector; by testing these classes, we can test all other classes of the system.

Tables 2 and 3 shows the test cases for each function and the expected result. The tests cases for the functions collect_issues, collect_pulls, collect_comments, collect_commits, collect_events, and, collect_commits_on_pulls consisted in two parts: (i) check the size of the list returned by these methods, and, (ii) checking if the id/hash/number of the last element returned by the API for the respective endpoint is correct. Concerning the function collect_all, the test case only checks part (i).

Furthermore, the tests cases for functions run_metrics and compile_data. Figures 7, 8, 9 show the test cases for both classes.

Table 2: Test suite for APICollector class.

Class	Function Name	Expected Result	Result	Passed?
APICollector	collect_issues	2	2	✓
APICollector	collect_issues	655043526	655043526	✓
APICollector	collect_pulls	1	1	✓
APICollector	collect_pulls	447649974	447649974	✓
APICollector	collect_commits	2	2	✓
APICollector	collect_commits	0263e46b9e4e6a50bf488de10 24b3e4b5bf121e6	0263e46b9e4e6a50bf488de10 24b3e4b5bf121e6	✓
APICollector	collect_comments	4	4	✓
APICollector	collect_comments	656896695	656896695	✓
APICollector	collect_events	2	2	✓
APICollector	collect_events	3536068866	3536068866	✓
APICollector	collect_all	2	2	✓
APICollector	collect_commits_on_pulls	1	1	✓
APICollector	collect_commits_on_pulls	18f2864b807dc50208b3f49d9 319ac1260499666	18f2864b807dc50208b3f49d9 319ac1260499666	✓

Table 3: Test suite for MetricsCollector class.

Class	Function Name	Expected Result	Result	Passed?
MetricsCollector	run_metrics	11	11	✓
MetricsCollector	run_metrics	3	3	✓
MetricsCollector	run_metrics	1	1	✓
MetricsCollector	compile_data	True	True	✓
MetricsCollector	compile_data	3	3	✓
MetricsCollector	compile_data	'1'	'1'	✓

```python
class TestAPICollector(unittest.TestCase):

    def test_collect_issues(self):
        """
        Testing the function collect_issues from APICollector class
        First test case sees if the list returned has size two, since there are just two issues on the repository (one issue and one pull request)
        Second test case sees if the id of the first issue is 655043526, since this is the ID returned for the last issue of the repository
        """
        print('Testing collect issues')
        collector = APICollector()
        owner = 'guriosam'
        project = 'test_collector'
        assert (len(collector.collect_issues(owner, project)) == 2)
        assert (collector.collect_issues(owner, project)[0]['id'] == 655043526)

    def test_collect_pulls(self):
        """
        Testing the function collect_pulls from APICollector class
        First test case sees if the list returned has size one, since there are just one pull request on the repository
        Second test case sees if the id of the first issue is 447649974, since this is the ID returned for the pull request on the repository
        """
        print('Testing collect pulls')
        self.collector = APICollector()
        owner = 'guriosam'
        project = 'test_collector'
        assert (len(
            self.collector.collect_all(project, PullsAPI(owner, project), 'pulls', 'number', PullRequestDAO())) == 1)

        assert (self.collector.collect_all(project, PullsAPI(owner, project), 'pulls', 'number', PullRequestDAO())[0][
                    'id'] == 447649974)

    def test_collect_commits(self):
        """
        Testing the function collect_commits from APICollector class
        First test case sees if the list returned has size two, since there are just two commits on the repository
```

Figure 7: Tests of API Collector class.

```python
class TestMetricsCollector(unittest.TestCase):

    def test_run_metrics(self):
        """
        Tests the run_metrics function of the MetricsCollector class
        First test here analyzes if each metric is generating a CSV in the end.
        In order to do that, we check if a list with the length of each CSV have the size of the input list of metrics.
        Total metrics = 11
        Second test analyzes if the length of the 'number of users' metric is 3.
        Third test analyzes if the length of the 'discussion length' metric is 1.
        """
        project = 'test_collector'
        collector = MetricsCollector('', project)
        assert (len(collector.run_metrics()) == 11)
        assert (collector.run_metrics()[0] == 3)
        assert (collector.run_metrics()[10] == 1)

    def test_compile_data(self):
        """
        Tests the compile_data function of the MetricsCollector class
        First test analyzes if the CSV returned is not NULL
        Second test analyzes if the length in lines of the CSV is equal to three (header plus two rows)
        Third test analyzes if the identifier of the second row is 1, since is the identifier of the first issue.
        """
        project = 'test_collector'
        config = JSONHandler('../').open_json('config.json')
        collector = MetricsCollector(config['output_path'], project)
        assert (collector.compile_data() is not None)
        assert (len(collector.compile_data()) == 3)
        assert (collector.compile_data()[1][0] == '1')
```

Figure 8: Tests of Metrics Collector class.

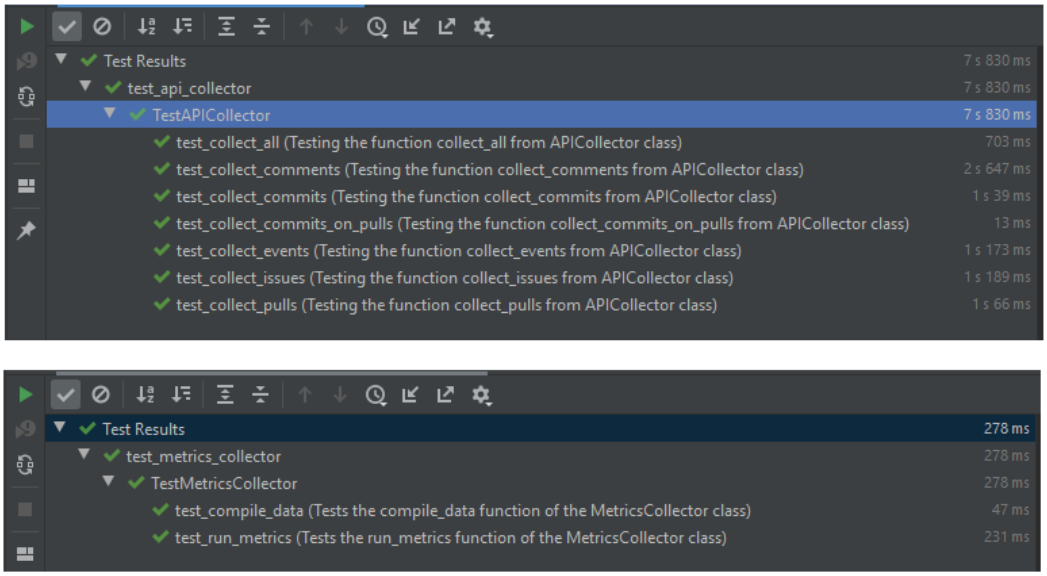

Figure 9: Tests suites

6 Installation Manual

This section presents the installation manual and the use guide for the GHFramework.

6.1 User Guide

1. Clone the GHFramework from the following GitHub Repository: `https://github.com/guriosam/GHPyFramework`;
2. Import the projectr to your python IDE. (PyCharm is recommended)
3. Change information in the config.json in the source folder. This file needs your GitHub Username, your GitHub Auth Token (can be generated at `https://github.com/settings/tokens`, the Output Path and the information (owner and repository name) about the projects you want to work with.
4. Instantiate the APICollector class and run the methods of the data you need to collect;
5. Instantiate the MetricsCollector class after the APICollector finished and call the *run_metrics* method passing as parameter the list of metrics you want to execute (empty list run all metrics).

There is a main.py class on the root of the project with a running example of the framework.

7 Final Remarks

This work introduced a framework called GHFramework, which is used to facilitate the work of researchers that want to collect and analyze GitHub's repositories. This framework uses the GitHub API (Section 2) to collect the information of the repositories. As future work, we intent to increase the number of endpoints supported and, also, a the number of metrics available.

References

[1] GITHUB, .. **The state of the octoverse**. `https://octoverse.github.com`, 2018. [Online; accessed 11-Jan-2019].

[2] EYOLFSON, J.; TAN, L. ; LAM, P.. **Do time of day and developer experience affect commit bugginess?** In: PROCEEDINGS OF THE 8TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 153–162. ACM, 2011.

[3] ŚLIWERSKI, J.; ZIMMERMANN, T. ; ZELLER, A.. **When do changes induce fixes?** In: ACM SIGSOFT SOFTWARE ENGINEERING NOTES, volumen 30, p. 1–5. ACM, 2005.

[4] RAHMAN, F.; DEVANBU, P.. **Ownership, experience and defects: a fine-grained study of authorship**. In: PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 491–500. ACM, 2011.

[5] TUFANO, M.; BAVOTA, G.; POSHYVANYK, D.; DI PENTA, M.; OLIVETO, R. ; DE LUCIA, A.. **An empirical study on developer-related factors characterizing fix-inducing commits**. Journal of Software: Evolution and Process, 29(1), 2017.

[6] HATTORI, L. P.; LANZA, M.. **On the nature of commits**. In: PROCEEDINGS OF THE 23RD IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, p. III–63. IEEE Press, 2008.

[7] MUTHUKUMARAN, K.; CHOUDHARY, A. ; MURTHY, N. B.. **Mining github for novel change metrics to predict buggy files in software systems**. In: 2015 INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE & NETWORKS (CINE), p. 15–20. IEEE, 2015.

[8] FALCÃO, F.; BARBOSA, C.; FONSECA, B.; GARCIA, A.; RIBEIRO, M. ; SALES, F.. **On relating technical, social factors, and the introduction of bugs**. arXiv preprint arXiv:1811.01918, 2018.

[9] ANICHE, M.. **Repodriller software**. `https://github.com/mauricioaniche/repodriller`, 2012. [Online; accessed 11-Jan-2019].

[10] SPADINI, D.; ANICHE, M. ; BACCHELLI, A.. **Pydriller: Python framework for mining software repositories**. In: PROCEEDINGS OF THE 2018 26TH ACM JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2018, p. 908–911, New York, NY, USA, 2018. ACM.

[11] GOUSIOS, G.; SPINELLIS, D.. **Ghtorrent: Github's data from a firehose**. In: MINING SOFTWARE REPOSITORIES (MSR), 2012 9TH IEEE WORKING CONFERENCE ON, p. 12–21. IEEE, 2012.