WikiContrib

Release 2.0

Jul 09, 2022

Contents:

CHAPTER 1

Intro

WikiContrib is a contribution data visualization tool which can be used to view a developer's contributions from different contributing platforms to the Wikimedia Foundation. Wikimedia Scholarship Committee can use this tool to decide who gets scholarship to Wikimedia events, and Wikimedians can also share their contributions profile with peers and employers.

Currently, it fetches the contributions from

1. Gerrit
2. Phabricator
3. Github

It visualizes the data in form of graphs and user contribution calendar. The tool can also be used by anyone to view the contributions of any contributor in the Wikimedia Foundation.

Note: Please note that all functionalities associated with the features allowing users to add multiple contributors details at once has been disabled. If these features are important to you, you can reach out on Github or WikiContrib talk page

CHAPTER 2

Usage

As specified, WikiContrib is used in visualizing the contributions of Wikimedians in the form of graphs and user contribution calendar. Let's start the discussion with how to use the tool first.

2.1 Intro to Queries

A Query is initiated whenever a user searches for the contributions of a single or a group of Wikimedians. Each query is uniquely identified by a user-friendly hash. Each query has a **group of filters** associated with it. There are two filters in a query:

1. From time.
2. Time Range.

From time: The user can get all the commits from a specified time (i.e month and year).

Time Range: From the specified From time, the user needs to give a range of time to which he/she want to fetch the details with the maximum being one year (i.e last one year, last one month or last 6 months, etc).

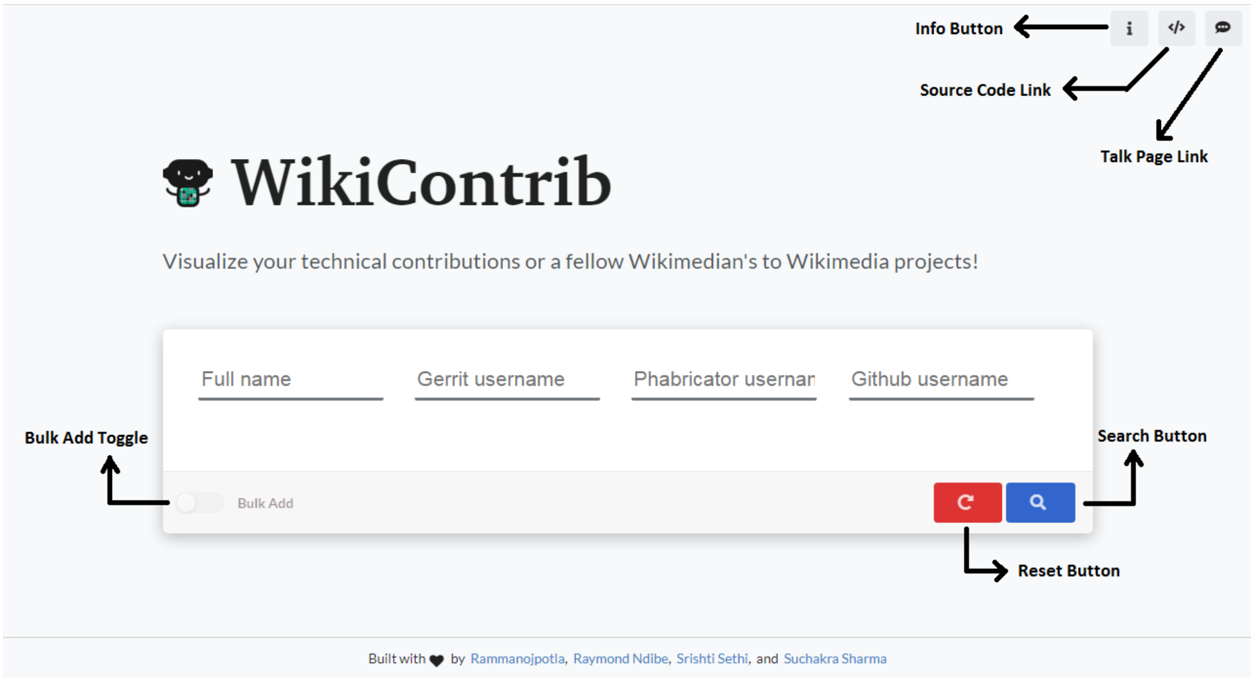
2.2 Query Creation

To create a query, the user needs to add the data (contributors full name, usernames of Gerrit, Phabricator, and Github) to the query. The user can add the data in two different ways:

1. Entering usernames manually.
2. Adding usernames in bulk.

Entering usernames manually:

Home page


Info Button
Source Code Link
Talk Page Link
WikiContrib
Visualize your technical contributions or a fellow Wikimedian's to Wikimedia projects!
Full name
Gerrit username
Phabricator usernar
Github username
Bulk Add Toggle
Bulk Add
Search Button
Reset Button
Built with by Rammanojpotla, Raymond Ndibe, Srishti Sethi, and Suchakra Sharma

Whenever the user goes to the home page, the above page will be displayed. In the above, the table has four columns. They are:

1. Fullname
2. Gerrit Username
3. Phabricator Username
4. Github Username

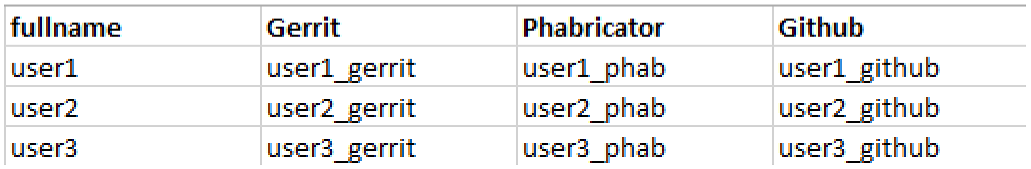
The user needs to fill the above four details of the corresponding Wikimedian whom the user wants to view the contributions of. `Gerrit Username` is used to fetch the data from **Gerrit APIs**, same with `Phabricator` and `Github` usernames. The details fetched from the APIs are associated with a common name i.e `Fullname` of the user. Fullname is also used in searching the users, we will discuss the searching in the later part of the doc.

There is an option provided to add usernames of multiple Wikimedians. By clicking on **Add Row** button, another empty row is added to the DOM. Similarly, the user can add any number of rows and fill the usernames into them. One of the cool things about the tool is the usernames the user entered will be cached in the user device cache. So, even if the user refreshes or closes the page, the details will not be lost! the user can fill a few usernames at some time, close the page and re-open it at some other time and add few other usernames. The user can also clear the cached data. Clicking the **Reset** button clears all the data he/she entered into the tool.

Adding Usernames in Bulk:

Entering usernames of tens of users is easy. But what if the user wants to know the contributions of hundreds of **Wikimedians**? It will take a lot of time and work to fill them manually to the tool. So, there is also an option to upload all the usernames in a CSV file. On clicking the toggle bar with text “Bulk Add”, the user will be provided with an option to upload a CSV file. If the user is uploading a CSV file, he/she needs to fill the data in the file in a particular format.

The CSV format is:

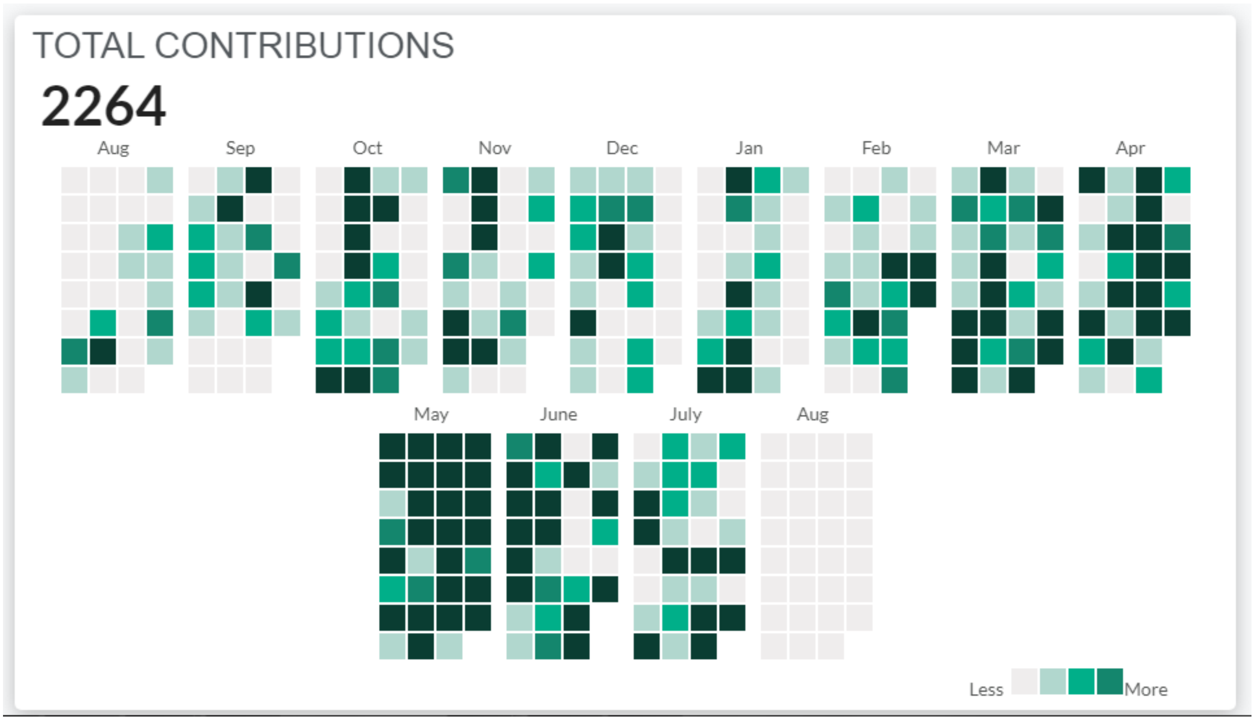


fullname	Gerrit	Phabricator	Github
user1	user1_gerrit	user1_phab	user1_github
user2	user2_gerrit	user2_phab	user2_github
user3	user3_gerrit	user3_phab	user3_github

Once the data is provided, (either entered manually or using a CSV file), The user can click the **search** button and this initiates a request to the server. The tool first verifies that the usernames provided belong to the same user and if not, warns the user of the mismatch. The user can decide whether to proceed or to crosscheck the provided usernames. If the usernames match or if the user decides to proceed, the tool then makes API requests to Gerrit, Phabricator, Github APIs. Once all the required details are fetched, it redirects to a URL `/<query_hash_code>`. Now the user can see the graphs of user contributions vs time along with a calendar that displays the contributions.

Note: `query_hash_code` is the hash-code generated by the query, the query can be accessed at any point in time using the hash-code.

User contribution calendar looks like:



If the user clicks on a specific date in the above calendar, all the commits made by the Wikimedian along with the `platform` will be displayed.

Hovering on the **info** button gives a popup with an intro paragraph about the tool.

2.3 Viewing / Updating filters

The user can view the results by following the above process of creating a query, there are also few filters displayed along with the result. The filters can be updated. Updating the current filters performs an API request and fetches the contributions of the Wikimedian according to the filters the user provided.

There is also an option to reset the filters to the default ones. Filters are associated with the Query. The contributions of all the Wikimedians are fetched according to the filters the user changed!

Note: Presently, the user can see all the contributions of any Wikimedian for the past one year (at maximum).

2.4 Viewing results

Once the contributions of the user are fetched, these things are displayed:

1. Graph of user contributions in **Gerrit**.
2. Graph of user contributions in **Phabricator**.
3. Graph of user contributions in *Github*.
4. A simple calendar that displays all the user contributions for the period you provide (similar to **Github green squires**).

At a time, the contributions of a single user are displayed. There are arrows provided to get the details of the next and previous user to the current user. There is also an input box provided. If you want to get the contributions of a specific user, you can search the `fullname` of the user in the search box. It displays the recommendations of the top 50 matching users.

2.5 Updating Queries

Once a user creates a query with the usernames of a set of Wikimedians and at a later point of time, if he/she wants to know the contributions of another Wikimedian, instead of creating a new query for a single Wikimedian, he/she can update the query and add the corresponding usernames.

There are four main different types of updates possible:

1. Initially a **CSV file** can be provided, another **CSV file** can be provided while updating the query.
2. Initially a **CSV file** can be provided, a set of **usernames of Wikimedians** can be provided manually while updating the query.
3. Initially a **set of usernames** of Wikimedians are provided manually, a **CSV file** can be provided while updating the query.
4. Initially a **set of usernames** of Wikimedians are provided manually, another set of **usernames of Wikimedians** are provided manually while updating the query.

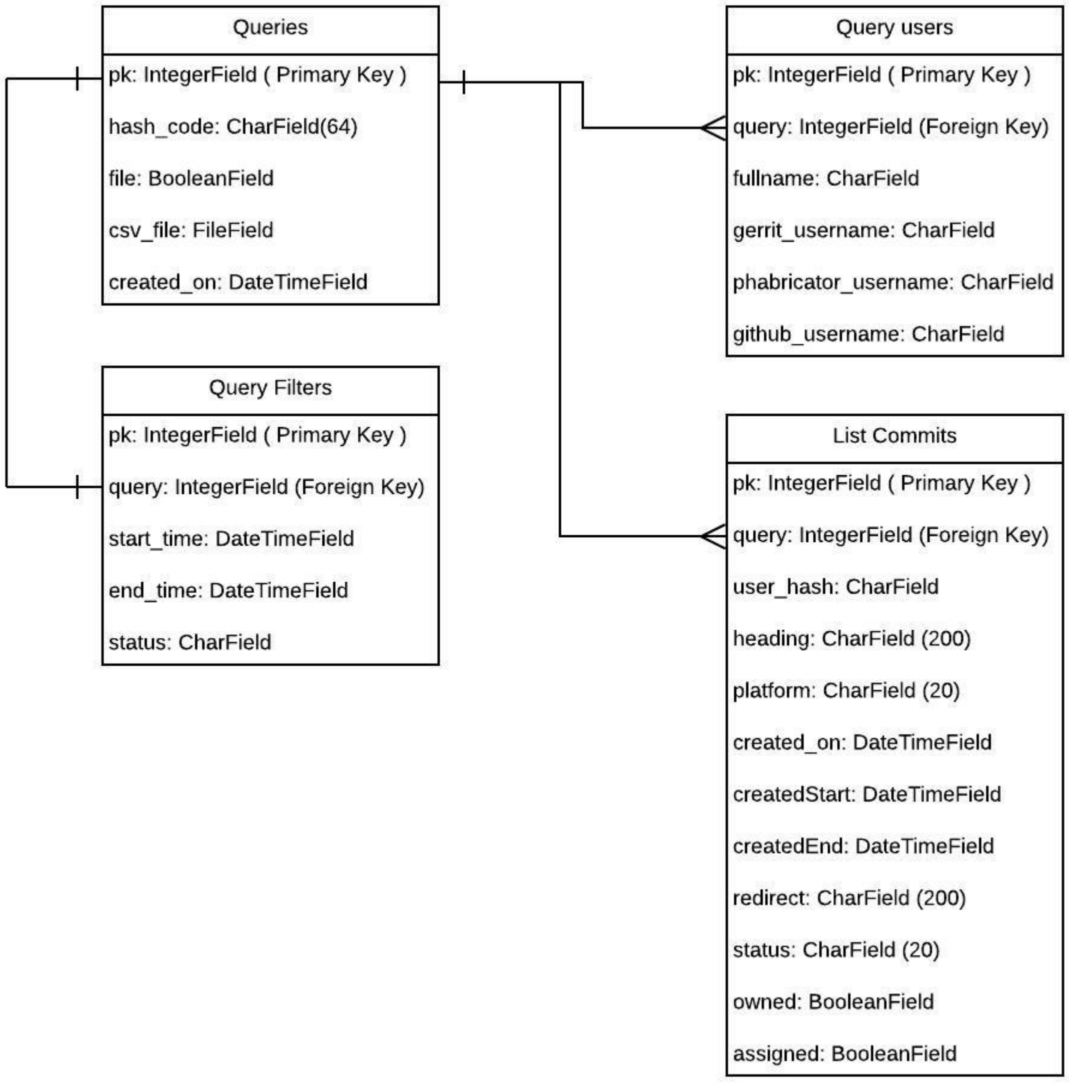
Note: Please note that all functionalities associated with the features allowing users to add multiple contributors details at once has been disabled. If these features are important to you, you can reach out on Github or WikiContrib talk page

CHAPTER 3

Internal Working

In the **Usage** section, we discussed the architecture and how to use the tool. Let's extend the discussion with a complete note of how the tool works internally.

We shall start our discussion with the schema diagram of the tool's database.


Queries
pk: IntegerField (Primary Key)
hash_code: CharField(64)
file: BooleanField
csv_file: FileField
created_on: DateTimeField
Query users
pk: IntegerField (Primary Key)
query: IntegerField (Foreign Key)
fullname: CharField
gerrit_username: CharField
phabricator_username: CharField
github_username: CharField
Query Filters
pk: IntegerField (Primary Key)
query: IntegerField (Foreign Key)
start_time: DateTimeField
end_time: DateTimeField
status: CharField
List Commits
pk: IntegerField (Primary Key)
query: IntegerField (Foreign Key)
user_hash: CharField
heading: CharField (200)
platform: CharField (20)
created_on: DateTimeField
createdStart: DateTimeField
createdEnd: DateTimeField
redirect: CharField (200)
status: CharField (20)
owned: BooleanField
assigned: BooleanField

As you can see above, there are four tables:

1. Query.
2. Query users.
3. Query Filters.
4. List commits.

Query has the data regarding the `hash`, `created time` etc. The usernames of the Wikimedians that the user provided during the query creation will be stored in **Query users** table. All the filters associated with the query will be stored in **Query Filters** table.

Whenever a user attempts to create or update a query (change the usernames), we first make sure that the usernames belong to the same user by calling `matchFullNames` view and returning a response that helps us decide whether to proceed or to first warn the user of the mismatch. Let's dig deep into the working of `matchFullNames` view.

This view uses `asyncio` and `aiohttp` to perform API requests in a parallel manner. It formats the submitted usernames, creates an event loop, and passes the event loop to `concurrent_get_fullnames` where we create three co-routines for Phabricator, Gerrit, and Github and attempts fetching the contributors fullnames on these platforms through `get_full_name`. (If you are not familiar with event loops and co-routines, they are used to perform threading programmatically, you can get more information about them here).

The above code adds three tasks to the event loop. Each of the tasks fetches the contributor's fullname on one of the three platforms. These are parallel because, let's assume there are two tasks **task1** and **task2**, initially, the loop started executing **task1**. If **task1** performs any API request, it has to wait till the response is received to proceed further. So, whenever the **task1** performs an API request, **asyncio** stores the state of **task1** and starts executing **task2**. When the response to the **task1** is received, it stores the current task and executes the **task1** further. When we are done attempting to fetch the fullnames, the results are passed to `fuzzyMatching` which uses fuzzy matching to figure out the probability that the usernames belong to the same user. the result of this matching is then returned as HTTP response.

If the usernames belong to the same user or if the user ignores the warning when usernames don't belong to the same user, we then query the server. Initially, a class named `AddQueryUser` view is triggered. The view creates a Query with a hash that is based on the usernames submitted and adds the provided usernames data to the query. This also creates a default set of filters and returns a redirect to `/<hash>` URL.

The URL triggers `DisplayResult` view. This view gets the necessary query data from the database and passes it on to `getDetails` where we decide whether to perform external API requests, fetch the details and cache the fetched data in the database or to return the already cached data from the database depending on whether we have cached that particular query in the past and if the query is not more than a day old. It also formats the data and returns the data to the browser as an HTTP response. Let's dig deep into the working of `DisplayResult` view.

This view also uses `asyncio` and `aiohttp` to perform API requests in a parallel manner. There are few constraints with the existing Phabricator, Gerrit, and Github APIs. Both Phabricator and Gerrit can not return the count of contributions made by a particular user. They will return the contributions made by the user in the form of a list of JSON objects. The good thing about `Gerrit` is it returns contributions of all the users with a single API request. But in the case of phabricator and Github, they will paginate the results with a max of 100 contributions on each page. For example, if a user performed 1000 different actions in phabricator or Github, then 10 API requests are to be made to get all the actions performed. Another constraint is that all the API requests for both Phabricator and Github are to be made sequentially. The API requests can not be parallel because each page has to be requested with a reference(except the first one). The reference to a page **n** will be provided on page **n-1**. Suppose if you have to get the commits of the user in the 7th page, you have to request the 6th page first to get the reference to the 7th page. To get the 6th page you have to request the 5th page and so on.

So, even if I want to get some page **n** you have to get all the details from **1 to n**.

In this tool, all the contributions of the user from Gerrit are being fetched. But in the case of phabricator, two kinds of tasks are taken into count:

1. Tasks authored by the user.
2. Tasks assigned to the user.

For Github, we are only concerned with the contributors commits.

DisplayResult view gets all the data required to perform the external API requests and calls another function `getDetails`. This function takes the data and formats it according to the requirement. It also creates a new **asyncio event loop**. This loop is first passed to `get_full_names` where we create three co-routines for Phabricator, Gerrit, and Github and attempts fetching the contributor's fullnames on these platforms through `get_full_name`.

When we are done attempting to fetch the fullnames, the results are passed to `fuzzyMatching` (just like in `matchFullNames`) which uses fuzzy matching to figure out the probability that the usernames belong to the same user.

After this, we call `get_cache_or_request` passing it several arguments some of which are query and the same event loop used to fetch the fullnames not long ago.

Inside `get_cache_or_request`, if the query exists in the cache and it is not older than one day, we fetch it from the cache and pass it to `format_data` where the data is properly formatted before finally returning it. If the query is not in the cache or is more than a day old in the cache, we call `get_data` where we create four co-routines to fetch the contributions data for the different platforms (two co-routines belong to Phabricator).

```
async def get_data(urls, request_data, loop, gerrit_response, phab_response,
                  github_response, phid, full_names):
    tasks = []
    async with ClientSession() as session:
        tasks.append(loop.create_task((get_gerrit_data(urls[1], session,
        gerrit_response))))
        tasks.append(loop.create_task((get_task_authors(urls[0], request_data
        , session, phab_response, phid))))
        tasks.append(loop.create_task((get_task_assigner(urls[0], request_data,
        session, phab_response))))
        tasks.append(loop.create_task((get_github_data(urls[2], request_data[3]
        , session, github_response, full_names))))
        await asyncio.gather(*tasks)
```

The above code adds four tasks to the event loop. Each of the tasks fetches contributions data through the various APIs.

1. `get_gerrit_data()`: fetch contributions user from gerrit.
2. `get_task_authors()`: fetch tasks authored by a user in phabricator.
3. `get_task_assigner()`: fetch tasks assigned to a user in phabricator.
4. `get_github_data()`: fetches contributions to a given set of Wikimedia repositories on github.

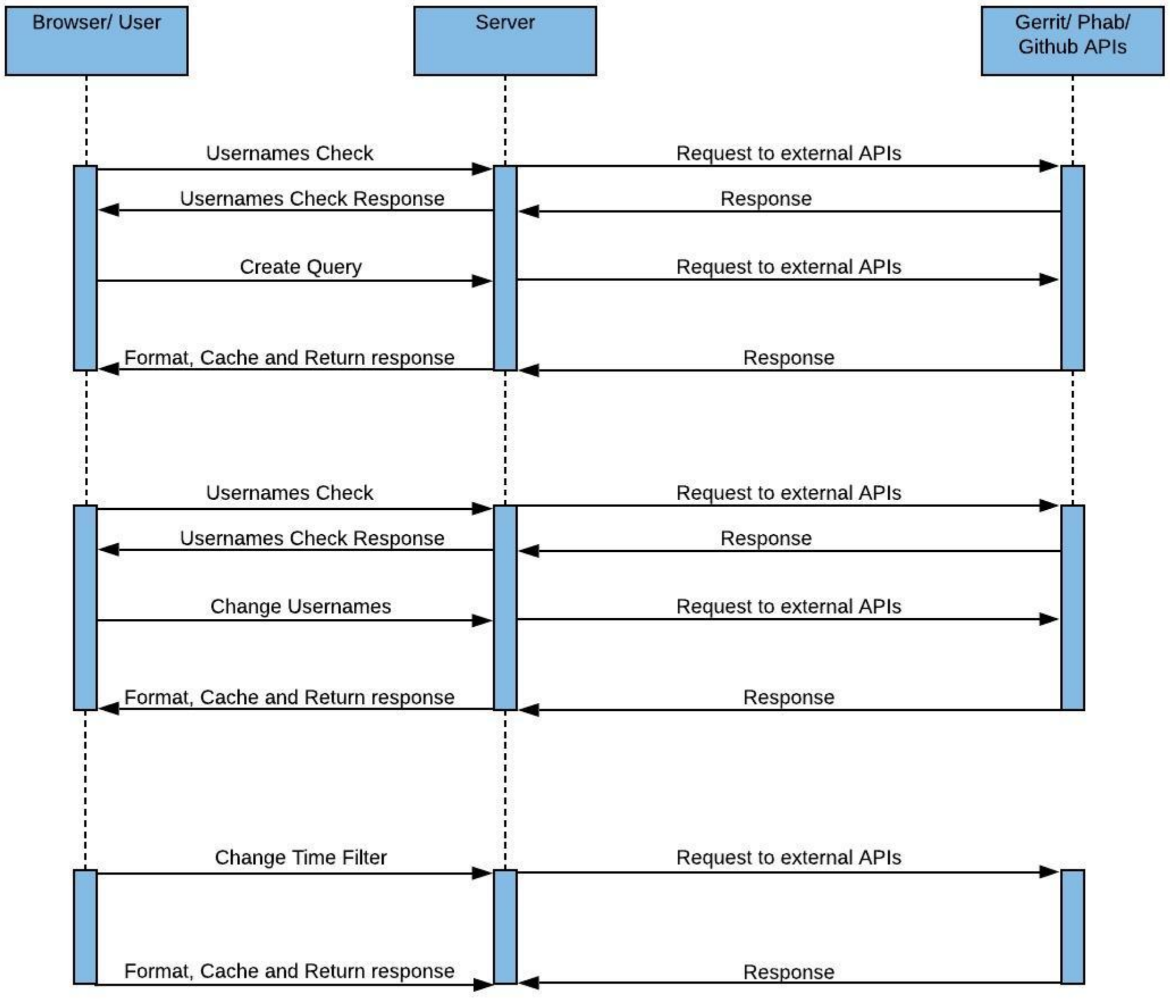
`get_gerrit_data()` perform a single API request and gets all the details of the users. `get_task_authors()` and `get_task_assigner()` gets the data but, as discussed above, phabricator APIs are paginated. So, these two co-routines have to request the data again and again, until there are no more pages left behind to request. “*get_github_data()* creates additional co-routines to get the contributions to a given set of repositories in a parallel manner.

Once the entire data are received, it is formatted and cached in the table `List Commits`. Apart from storing them in databases, the commits that meet the requirement of all the Query filters are taken and the response is returned to the user. For the sake of performance, the contributions of at the max. of past one year are being requested.

Whenever the filters or usernames of a query are changed, then these whole processes are repeated.

The view `GetUserCommits` returns all the commits of a user on a particular date.

sequence diagram:



If you want to know more about the tool, you can refer to the API documentation from here.

CHAPTER 4

Installation

Initially, clone the repo with the command.

```
git clone https://github.com/wikimedia/WikiContrib.git
```

The tool has two different components(Backend and Frontend). Each of them has its installation instructions.

4.1 Backend

Now, if you type command `ls` (for linux) or `dir` (for windows), you can see a directory named **WikiContrib**. Go inside the directory using the command `cd WikiContrib`. There will be two directories in it:

1. backend
2. frontend

If you go inside `backend` directory(use command `cd backend`). You can find another directory named `WikiContrib`. It is the main project backend directory. cd into `WikiContrib` and follow the instructions here to install the backend .

4.2 Frontend

You will see two directories in the project's root directory: 1. backend 2. frontend

If you go inside `frontend` directory(use command `cd frontend`). You can find another directory named `WikiContrib-Frontend`. It is the main project directory. cd into `WikiContrib-Frontend` and follow the instructions here to install the frontend .

CHAPTER 5

Contributing

5.1 Reporting bugs

If you find any bugs or issues while using the app. Feel free to report them here.

If you are reporting a new issue, please provide these details:

1. Title (Describe the entire issue in a single sentence).
2. Description (Write more about the issue).
3. How to reproduce the issue? Provide some screenshots (if possible).

5.2 Contributing to the repo

Thanks for choosing the project to contribute. You can install the repo locally through the installation instructions provided in the "Installation" section. You can find the list of issues in the tool here. You can choose an issue and fix it. Keep your master branch updated and pull all the changes before making a PR.

Note: If you are updating the backend after you make some change, you can check if all the tests are passing using the command:

```
python manage.py test
```