Large-scale and Multi-structured Databases

InstaFound: a crowdfunding social network

Year 2020/2021

Authors:
Giulio Federico
Luigi Di Meo
Gianluca Sirigu

GitHub address: https://github.com/InstaFound/InstaFound

Index

Paragraph 1

Introduction: the idea

The application that was developed is called *InstaFound*. It's a social network based on crowdfunding, so as well as in each of them users can publish a post, but here the post has a different meaning as it represents a campaign that can be "liked" with a donation and can be followed. Every campaign is organized by the same user who published it, can be supported by a team and can be in favor of a beneficiary. It's also defined by an image, a description and belongs to a particular category. Every campaign aims to reach a certain amount of money for his cause and relies on user's donations and followers, but also on his possible team, composed by other users of the application, whose role is to make the campaign more visible among the other users. A user can explore the platform home page by scrolling through the latest campaigns, filtering by categories if he wants, goes to his profile and see the campaigns he has created with their related powerful statistics or those he is following. Like any self-respecting social network is also possible to add user to his friends list. The friends list is the list of people that can be chosen when a user creates a team to support his new campaign if he doesn't already have a suitable one.

Paragraph 2

Functional and not functional requirements

Functional Requirements

In the following bullet list is summarized the functional requirements of the application:

- The platform allows to be navigated in two types of modes: normal and exploration.
- The use of the *normal mode* is reserved for registered users only.
- The use of *exploration mode* is possible for any type of user but only if the previous mode is not available at that time.
- Any registered user can:
  - explore the platform home page by scrolling through the latest campaigns, filtering by categories if he wants.
  - follow a campaign.
  - make a donation to a campaign.
  - be suggested of any new friendships depending on where he lives, interests on the categories in common.
  - create his own campaign, choosing the support team from those he already created, the category, a description, an image, the amount to reach and possibly the beneficiary. When a user creates a campaign becomes an organizer.
  - No campaigns or accounts created can be deleted in order to maintain the traceability of operations.
  - open the details of a campaign to see the title, the image, the organizer and his support team,the complete list of donors with their contribution and timestamp, the total to be achieved and that achieved. If he is the organizer of the campaign he is seeing the details of, then he can also see the associated powerful statistics.
  - add any user he met as an organizer, team member or donor to his friends list.
  - edit his profile image.
  - see the campaigns he has created, the ones he is following, the list of created teams or those to which it simply belongs and finally see the complete list of friends.
  - create a new team and add users he is friends with as members.

    - see the teams and other campaigns supported by his team members.
    - delete a team if is the creator of that team.
    - remove a member of a team if is the creatori of that team
    - leave a team. If the user is the organizer this will also imply the elimination of the team.
    - remove a friendship.

No Functional Requirements

- The platform must only work in Italy.
- The platform must have elegant, minimal graphics and must be intuitive, easily usable by the user.
- For this reason the application must not collect other user data than those entered during registration.
- Campaign authors (organizer and support team members) must remain visible even if the team is canceled.
- The user interface must never stall during any operation except for those for which it is necessary to happen.
    - The donation operation must be completed on all servers before giving control back to the user.
- The application backend must rely solely on NoSQL technologies.
- The platform must be able to manage large volumes of data and be able to respond quickly even in these cases.
- The platform must be scalable.
- The platform must be reliable and continue to work even in case of network partitions.

Paragraph 3

Actors and Use Case Diagram

Given the requirements described above, the application will require the presence of 3 actors:

	Actors and granted operations
Unregistered user	Can only login or register on the platform (or use the *exploration mode* in case *normal mode* is not available)
User	After a login a *unregistered user* becomes *user* and so he can do almost anything, in particular he can browse a campaign finding it possibly filtering by category and once found he can donate, follow, show the entire list of donors, organizer and members of support team with the possibility to add each of them to own friends list. He can edit his profile changing his image. He can explore the list of created teams and decide whether to delete/add a new member or delete the whole team or simply see the teams he belongs to with the only possibility to leave it. Another list he can manage is that of friends in which he can also decide to delete one or more of them. An important operation granted is that of creating a campaign (choosing a title, an image, a description, the category, the amount to achieve) specifying any support team (choosing from those already created) and beneficiary. Finally he can logout.
Organizer	A *user* becomes an *organizer* when he creates his first campaign. He enjoys special actions only on his campaigns, such as seeing their statistics.

Graphically the Use Case Diagram is represented in *Figure 1*:

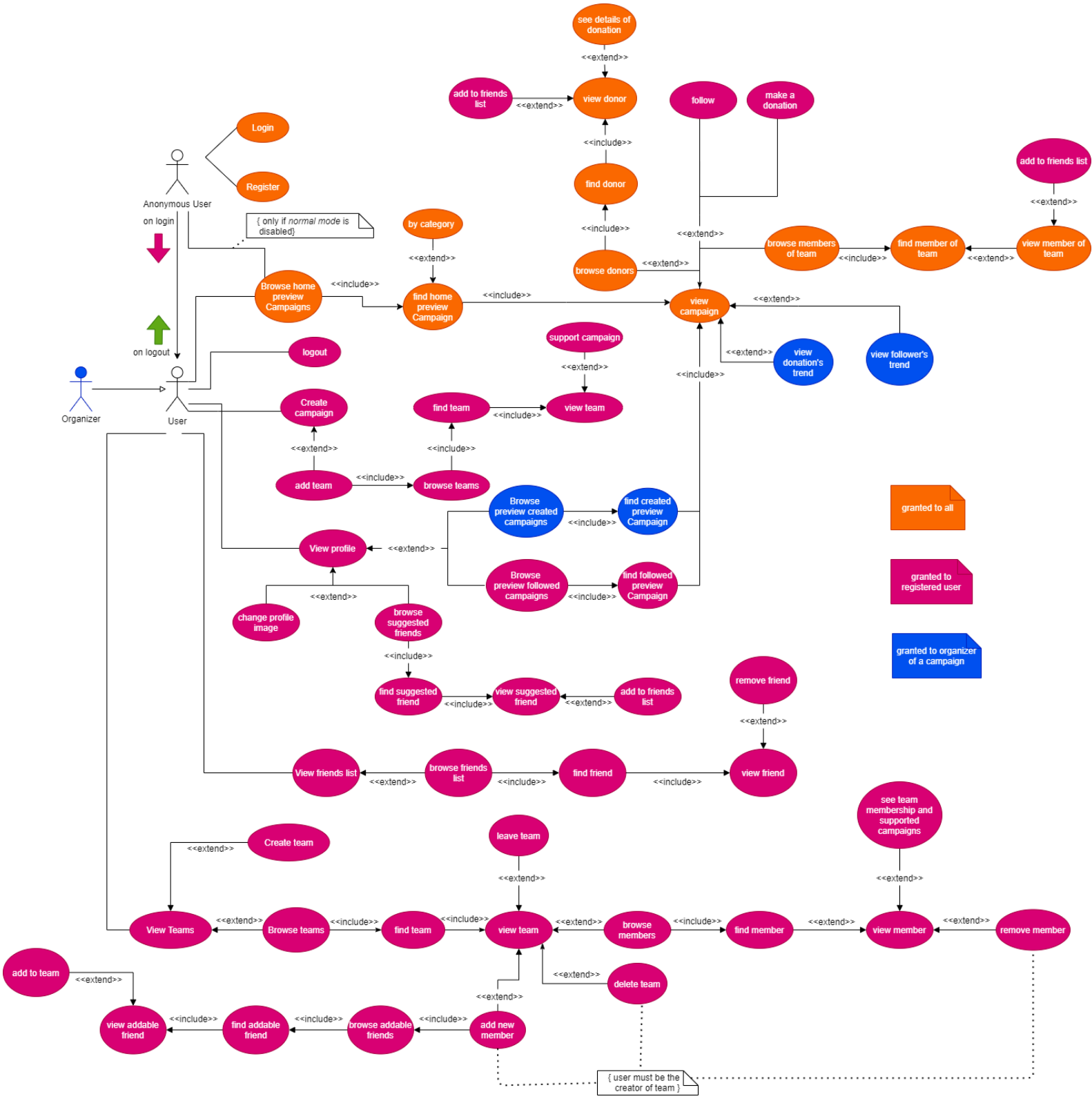


Figure 1: use case diagram.

Paragraph 4

Class Diagram

To implement the functional requirements and respect the non-functional ones, the following class diagram in *Figure 2* was chosen for the application:

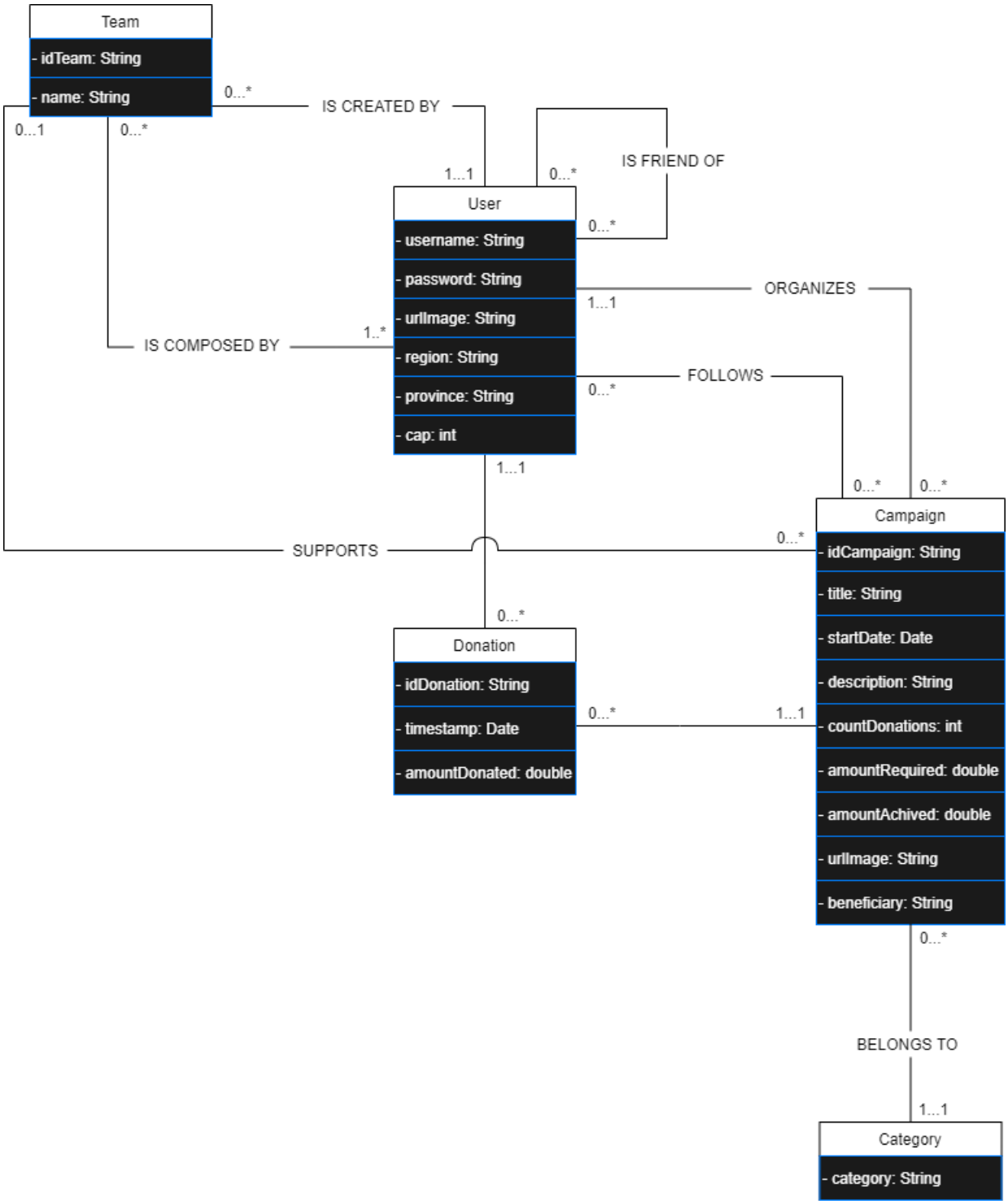


Figure 2: class diagram.

Below we define the meaning of each class and its attributes:

Class User		
Description		
A user contains all the information related to each user such as the teams he belongs to or who has created, the list of friends, the campaigns created, followed and to which he donated.		
Attributes	**Type**	**Meaning**
username	String	A name that uniquely identifies each user within the platform.
password	String	An alphanumeric string with which the user authenticates to access the portal.
userImage	String	The url where the user's current profile picture resides.
region	String	The region to which the user belongs.
province	String	The province to which the user belongs.
cap	int	Identifier of the city of the user.

Class Campaign		
Description		
A campaign is a post published by an organizer and keeps track of its information since its opening.		
Attributes	**Type**	**Meaning**
idCampaign	String	A identifier that uniquely identifies each campaign within the portal.
title	String	The title of the campaign.
startDate	Date	The publication date.
description	String	The description of the campaign.
countDonations	Int	The number of donations achieved.
amountRequired	Int	The number of money required to consider the campaign successful.
amountAchived	Int	The number of money achieved.
urlImage	String	The url where the campaign's picture resides.
beneficiary	String	the name of the beneficiary, if any.

Class Team		
Description		
A team is a group of users who are committed to supporting the campaign by giving it as much visibility as possible.		
Attributes	**Type**	**Meaning**
idTeam	String	A identifier that uniquely identifies each team within the portal.
name	String	The name of the team.

Class Donation		
Description		
A donation represents the contribution that each user makes to a campaign.		
Attributes	**Type**	**Meaning**
idDonation	String	A identifier that uniquely identifies each donation within the portal.
timestamp	String	The timestamp related to when the donation was made.
amountDonated	double	The amount in terms of money of the user's contribution.

Class Category		
Description		
A category is a type to which each campaign belongs.		
Attributes	**Type**	**Meaning**
category	String	A name that uniquely identifies the category.

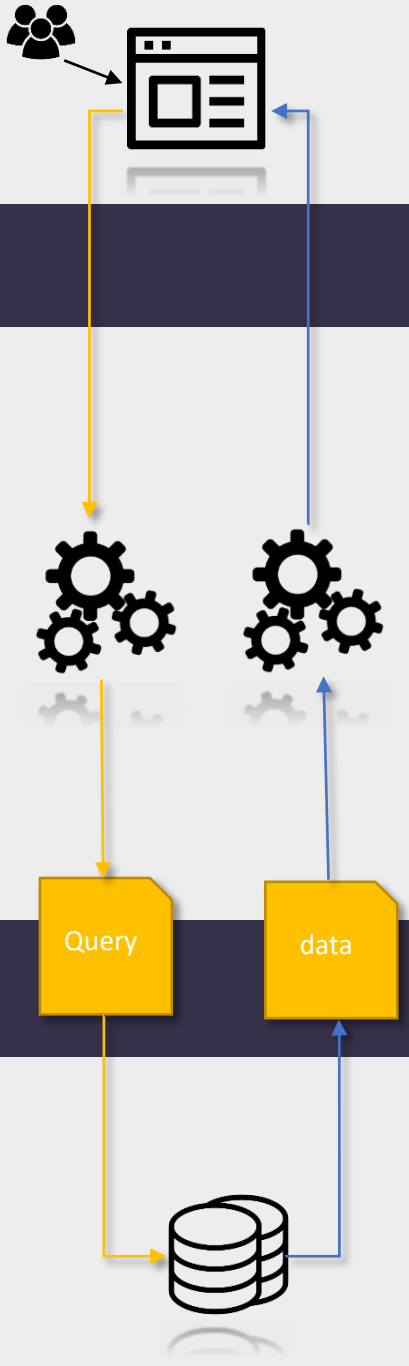
Paragraph 5

Architectural Design

For the software creation process we combine three separate architectural tiers known as *Front-End, Middleware* and *Back-End.*

Front-End tier

To implement the top-most level of the application javafx version 8 was used in order to allow user a friendly interface to access the platform.



Middleware tier

All the logic of the application is managed by this level which acts as a glue between the two levels of front-end and back-end.
The main modules that make up this level are the one related to the management of the database of the campaign and it's donation whose data are mainly stored on MongoDB with which we will exchange data through *MongoDB connector* and finally the module related to the main database management of users and their teams with which we will exchange data through *Graph connector*.

Back-End tier

MongoDB was used as the server side, which will mainly manage the campaigns end it's donations data, and Neo4J, which will mainly manage the data of users and their teams. Small data relating to campaigns are also managed by Neo4J in order to improve the user experience.

In particular, we deployed the database on three virtual machines made available by the University of Pisa. With these resources we will deploy a three member MongoDB replica sets and a single instance of Neo4j.

Visually the organization of the distribution is shown in the following *Figure 3*:

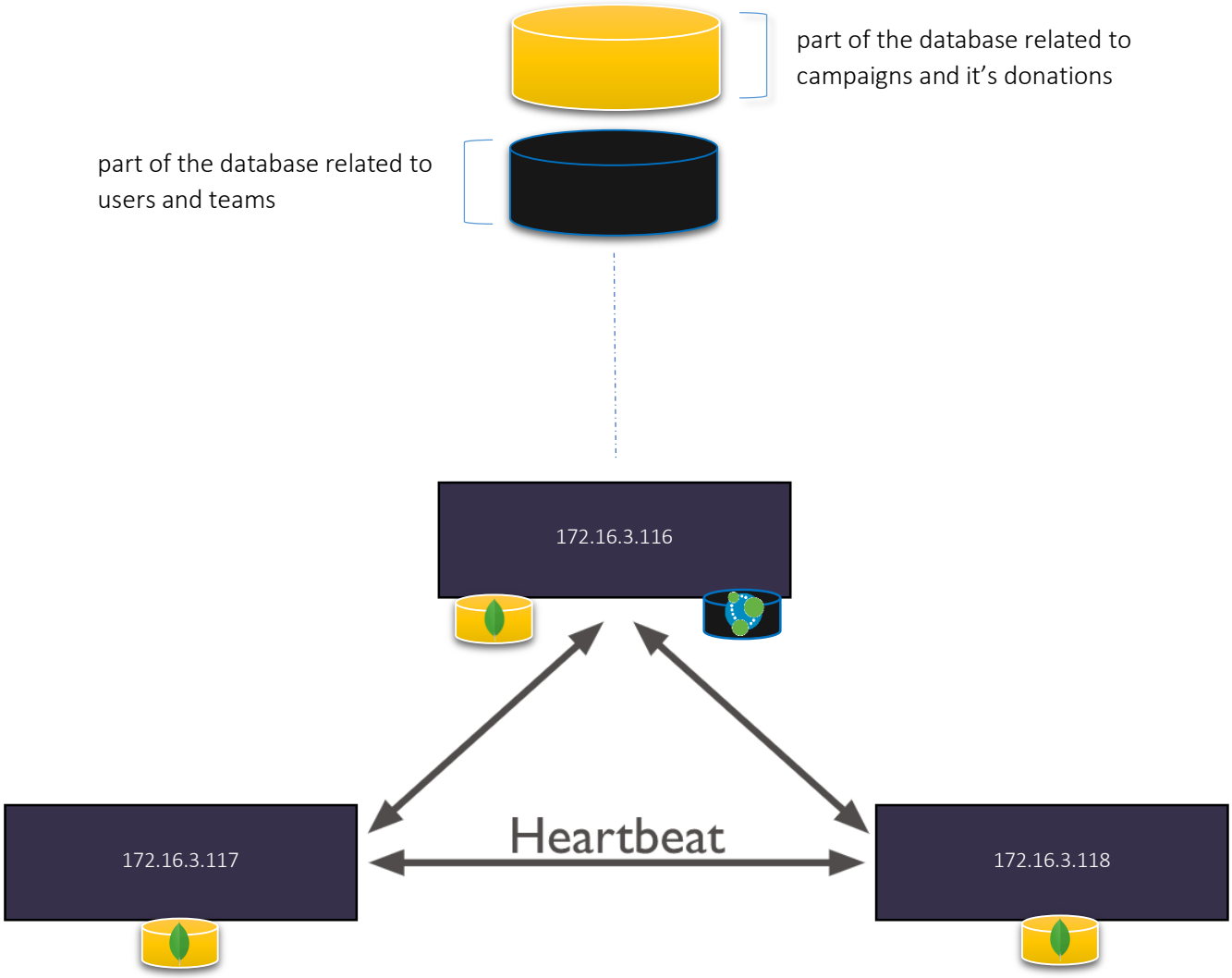


Figure 3: abstraction of Instafound master slave configuration.

The reason and how the data are divided in this way will be clearer from the next paragraph.

Paragraph 6

Data Organization

The reason the database was "split in two" is for purely engineering reasons. The application uses a MongoDB replica set consisting of three members in order to enforce persistence and availability even if consistency comes down because the application leads users to read to the replicas causing a "eventual consistency" which is not a big problem in a social network like this. For example, it does not matter if the user for a certain moment sees a campaign that has not updated the number of donations, or at least it is very convenient to give up this for greater availability, which is vital in dynamic platforms such as social networks. However, there will be some actions such as donations that will override this default setting and it will be expected that all donations in all the replicas will be committed. But let's cut to the chase. Operations such as donations, due to their sensitivity, will be distributed over the three replicas, as well as the creation and storage of the campaign status. The most dynamic actions, those that require greater navigation between the same or the different entities are managed by Neo4J whose single server certainly does not allow high availability but ensures a strong consistency. However, the non-high availability is not a very serious problem since the database will be interrogated by queries that will fully exploit its structure. Furthermore, the use of indexes (in both dbms) will accelerate slower queries even more at the cost of greater redundancy and write latency, the latter however will affect infrequent writes. The "piece" of dataset that MongoDB manages in common with Neo4J concerns two simple fields of the Campaigns. Surely this leads us to add redundancies and to "risk" in terms of consistency as we are accessing different databases, but first what is shared are small fields which are almost never altered and second we have adopted a strategy to ensure cross consistency (see paragraph 12). This small redundancy, however, allows us to exploit the Neo4j structure to execute certain queries that would be costly on MongoDB at the computational level.

In the *Figure 4* we can see how the class fields have been organized:

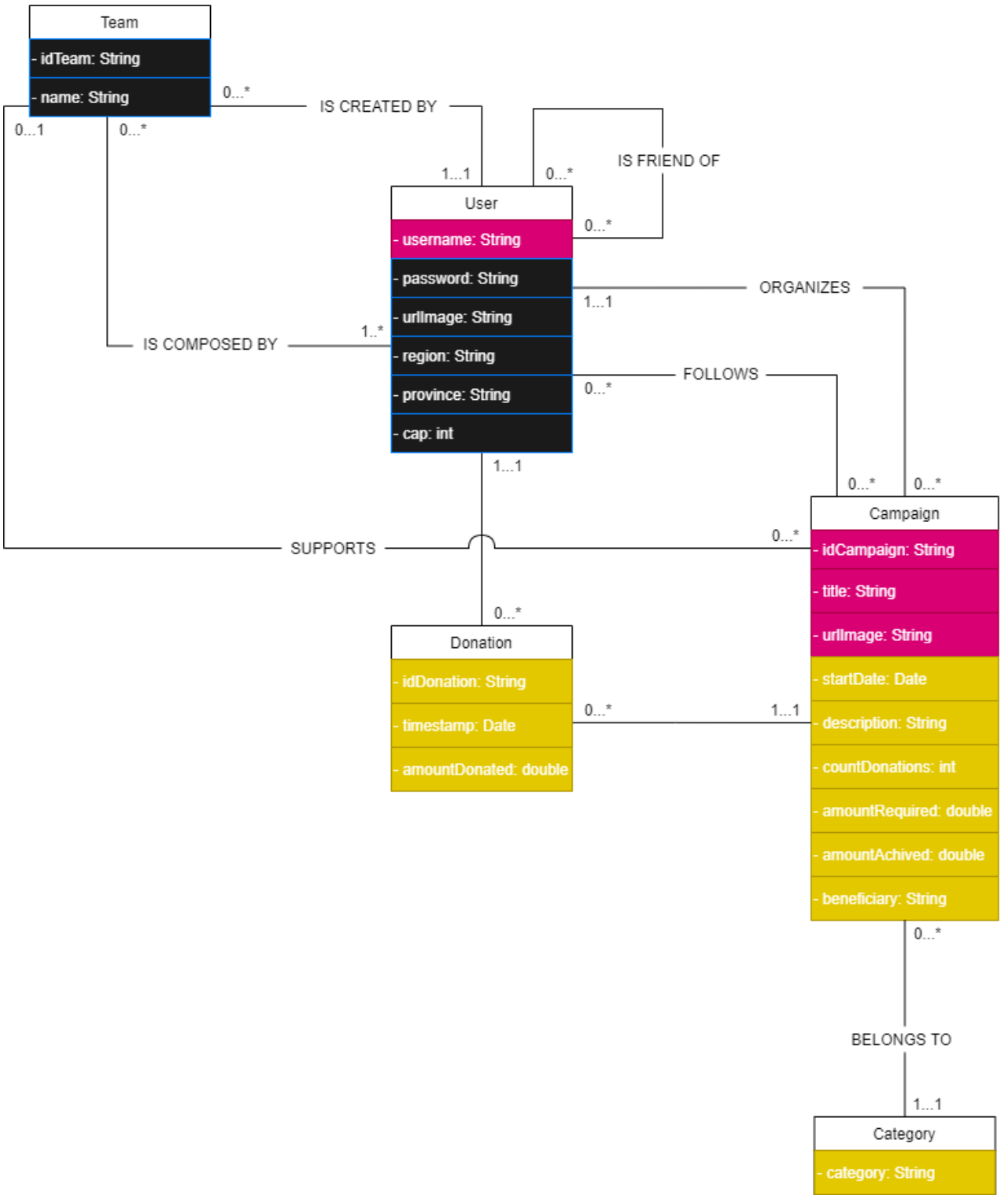


Figure 4: fields of interest of the two databases.

Paragraph 7

Logic behind the Data Organization

We can see the logical architecture behind the organization of data which is precisely that of their extraction.

Every day the gofoundme.com site is **scraped**. For security reasons, we use a VPN (not the one granted by the University of Pisa) to convey our requests:

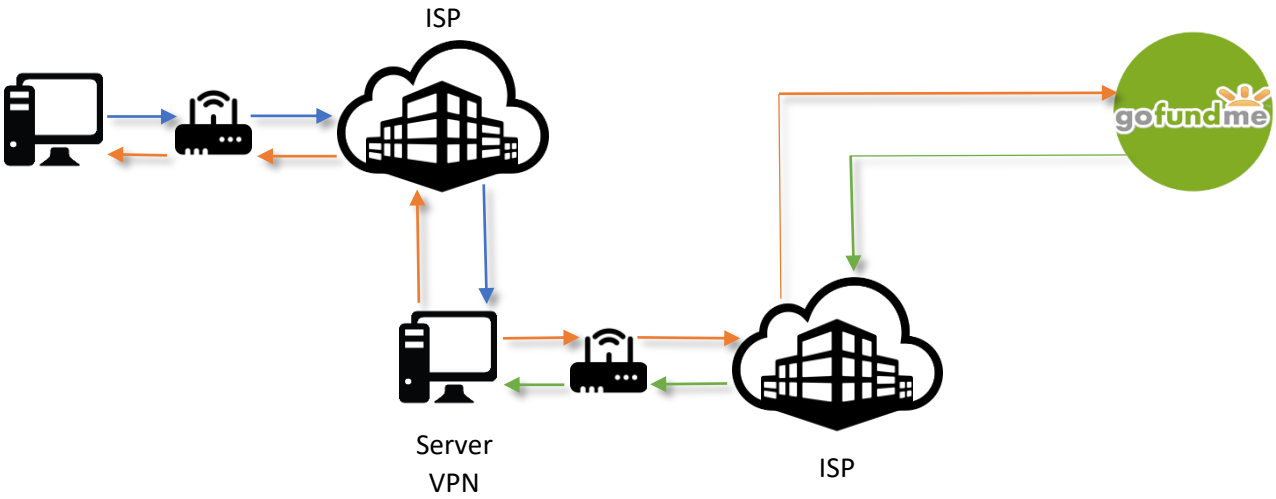


Figure 5: network scraping path.

Static content scraping is done through the **Jsoup parser** which connects to the site, downloads the page code and analyzes it. The scraping of *dynamic contents*, i.e. those reloaded via javascript, is done by simulating a browser through the *Chrome driver* using **Selenium**. But there are *particular data*, such as the entire list of donors and more precise numbers about the followers that can be obtained directly by manipulating the **http requests** and reading the payload of the response.

The raw data are saved in CSV format. When requested they will update the databases by converting them to JSON. As already said before, however, the data is divided into two parts, of which a third (very small) in common to the two databases:

- the *dataset of campaigns and donations* will contain all the information related to the campaigns and the respective donations made and will be managed only by MongoDB.
- the *dataset of users and their teams* will contain all information relating to users, their teams, friendships between users, participation and the creation of a team and will be managed only by Neo4J.

- the *dataset in common* to both will only see the "title" and “urlImage" fields of campaigns on Neo4J, but we will see that this will speed up performance.

Paragraph 8

MongoDB Design and Query Analysis

All the data of the campaigns and related donations have been compacted into a *single collection* composed by documents whose structure is in general as follows:


```
{
    “idCampaign”: “5b7d297cc718bc133212aa94”,
    “title”: “Aiutiamo la piccola Giulia”,
    “startDate”: “16/11/2020”,
    “description”: “Ciao a tutti sono un amico della famiglia che…”,
    “countDonors”: 1036,
    “amountRequired”: 50000,
    “amountAchived”: 21036,
    “urlImage”: “https://images.gofundme.com/IVdRkQZeaa...3692_r.jpeg”,
    “beneficiary”: “Alessandro Gervasio”,
    “category”: “Spese Mediche”,
    “donations”:
                [
                    {
                        “donorsUsername”: “federicaCostanzo”,
                        “amountDonated”: 10,
                        “timestamp”: “2020-12-05 09:30:16”
                    },
                    {
                        “donorsUsername”: “carloBonventre”,
                        “amountDonated”: 125,
                        “timestamp”: “2020-12-15 21:20:56”
                    },
                    {
                        “donorsUsername”: “carloBonventre”,
                        “amountDonated”: 125,
                        “timestamp”: “2020-12-15 21:20:56”
                    }
                ]
}
```

The reason why certain classes have been merged into a single class depends solely on the type and frequency of queries users execute on the system. MongoDB manages this dataset because it deals with more important and sensitive data than friendship or follow relationships. These data need to be replicated to safeguard them. Furthermore, the creation of campaigns are not so frequent as events, as well as donations, at least less frequent than actions such as follow, add a person to list of friends, leave a team. This is with regards to writing. However, by allowing users to read directly from the replicas, we restore the lost availability in writing. Furthermore, thanks to the use of the index, that we will see after, the less performing queries will be speeded up by decreasing latency and number of database accesses.

Compared to the original class diagram in Figure 2, the Category class has been incorporated as a field to avoid creating a separate collection due to the fact that it assumes limited values in the "category" field and does not contain any other information (no other fields) for which it is needed a separate collection.

The Donation class has also been incorporated as no query needs to manage all the donations on the site, rather every time the system will need the donations it will be only when it will have to show the entire list of donors and some statistics, but everything always related to each single campaign. There is just one query that draws up a ranking of the top donors, but in this case the solution is presented in the paragraph 9. However, there are some queries, such as those of reading the latest campaigns (possibly filtering by category) that are very frequent for which a separate collection of Donations or Category would perhaps have been to our advantage. However, if we had a Donations collection, we would in any case have to scroll through all its documents to retrieve those relating to a specific campaign. If we have a Category collection with an array field that contained all the ids of the historically created campaigns, we would certainly have easily managed the retrieval of the latest campaigns by category. In fact it is so, but we have let this behavior be automated by a specific compound index (see later) which certainly performs this task in a better way.

To perform the system and the queries it frequently receives, an additional support collection has been created which acts as a log for donations. This collection was initially conceived as Capped Collections but since the need was to eliminate documents older than one month (to draw up the ranking of the top monthly donors and top monthly categories) then a TTL index was used (see later).

```
{
    “username”: “carloBonventre”,
    “amount”: 125,
    “timestamp”: “2020-12-15 21:20:56”,
    “category”: “Spese Mediche”
}
```

Figure 6: model of document for *monthlyDonations* collections.

Let's see the main read and write queries handled by MongoDB.

Read operations		
Operation	Frequency	Cost
Show the N most recent campaigns with related information.	High	*Low
Filter all campaigns by category and sort them by the most recent.	High	*Low
For a specific campaign, retrieve the list of donors with related informations on how much and when they donated.	High	Low
Aggregation Retrieves the top donors of the last 30 days with their total donations.	High	*Medium
Aggregation Show the temporal trend of the daily average of donations for a specific own campaign.	Low	Medium

Aggregation Shows the ranking of the categories that received the most donations in the last 30 days.	Low	*Medium

The field values with (*) are costs that exploit the indexes (see paragraph 9).

Write operations		
Operation	Frequency	Cost
Create a campaign.	Medium	Low
Insert a new donation for a specific campaign.	Low	Low
Update the total number of donations for a specific campaign.	High	Low

The only implementations for aggregations are shown below:

Queries implementations

Retrieves the top donors of the last 30 days with their total donations.

```
db.monthlyDonations.aggregate
(
  [
    { $group: { _id: "$donor", totalDonated: { $sum: "$amount" } } },
    { $sort: {totalDonated: -1}}
  ]
)
```

For this query will be used a ***readPreference=SecondaryPreferred*** to speed up it and will be also use a ***readConcern = local*** to get the latest data that the server has, it doesn't matter if the roolback will be done due to the failure of the primary, because it is a rare event and then in a social network it is not a problem to show a ranking that is not completely reliable. The latest update is much more interesting.

Show the temporal trend of the daily average of donations for a specific own campaign.
(a dummy id of campaign will be used for example)

```
db.campaigns.aggregate
(
  [
    {$match: {_id: ObjectId("600c3a51dd23fe683fefaba7")}},
    {$unwind:"$donations"},
    {$group: {
                  _id: "$donations.timestamp",
                  avgDonations: { $avg: "$donations.amount" } }
                },
    {$sort: {_id: 1}}
  ]
```

)

For this query will be also used a ***readPreference=SecondaryPreferred*** and a ***readConcern = local*** for the same reasons mentioned above.

Shows the ranking of the categories that received the most donations in the current week.

```
db.monthlyDonations.aggregate
(
  [
    { $group: { _id: "$category", totalDonations: { $sum: "$amount" } } },
    { $sort: {totalDonations: -1}}
  ]
)
```

For this query will be also used a ***readPreference=SecondaryPreferred*** and a ***readConcern = local*** for the same reasons mentioned above.

Paragraph 9

MongoDB Index

The two collections created and the structure of each document they host was designed to make writing and reading operations as efficient as possible. The aggregations are also quite performing.

For greater performance, two different types of indices have been introduced:

- a *compound index* that aims to perform the filtering of campaigns by category and by the most recent.
- a *TTL index* that aims to improve the ranking of donors and categories.

Compound index

A compound index is defined on the values of the *category* and *startdate* fields of the campaign collection:

```
db.campaigns.createIndex( {"category": 1,"startDate": -1} )
```

This index contains static elements, i.e. elements that do not change over time minimizing one of the more negative sides of the indexes being updated every time. Thanks to the low frequency with which campaigns are created and the small part of them to be managed by the index, managing it does not affects performance.

The default index that MongoDB creates on the ObjectId index is not useful even if we wanted to find only the latest campaigns as MongoDB does not have a out-of-the-box auto-increment functionality like SQL databases, so ObjectId values do no represent a strict insertion order.

Below we can see the *computational cost* of loading the most recent campaigns of the Environment category **without index**:

```
db.campaigns.find({category: "Environment"}).sort({startDate:-1}).explain("executionStats")
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
            "inputStage" : {
                "stage" : "COLLSCAN",
                "filter" : {
        ...
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 109,
        "executionTimeMillis" : 4,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 751,
        "executionStages" : {
        ...
```

The COLLSCAN indicates that to perform the query it had to go through all 751 documents (totalDocExamined) to return the 109 documents (nReturned) desired. The totalKeyExamined at zero indicates the use of no index. Scanning all documents is always bad.

One might think that the index is no necessary as the user is only shown 10 campaigns per category at a time. Using "limit (10)" would not prevent from scrolling through all documents every time anyway. Furthermore, when the next 10 other campaigns are shown, a combination of "skip (10) .limit (10)" will always examine all documents, returning only to the end the next 10. This means that if it's necessary to load the campaigns from position 1000 to 1010 will have to load all and only at the end return the 10 documents.

Below we can see the *computational cost* of loading the most recent campaigns of the Environment category with index:

```
db.campaigns.find({category: "Environment"}).sort({startDate:-1}).explain("executionStats")
{
	"queryPlanner" : {
		"plannerVersion" : 1,
		 ...
			"inputStage" : {
				"stage" : "IXSCAN",
				"filter" : {
		 ...
	},
	"executionStats" : {
		"executionSuccess" : true,
		"nReturned" : 109,
		"executionTimeMillis" : 1,
		"totalKeysExamined" : 109,
		"totalDocsExamined" : 109,
		"executionStages" : {
		 ...
```

The IXSCAN indicates that to perform the query it uses the index, so it only need to go through all 109 documents to return the 109 documents desired. The totalKeyExamined at zero indicates that the query first accesses the index, performs a fast search through its structure to access the documents. Only the desired documents are consulted. Indeed, the time required (executionTimeMillis) was reduced by a factor of 4.

If it's necessary to load the other 10 documents, the search will be done always on 20 documents, but only 10 will be physically accessed.

A solution to avoid scrolling the index and also not create another index should be take the ObjectId of last loaded campaigns and then retrieve all the ten ObjectId greater than it. An index on ObjectId is automatically created by the system so will ben scrolled. But again, there are two problems: in any case it is necessary to physically access every single document to know if it belongs to that category, furthermore MongoDB does not guarantee that the ordering by ObjectId respects the time of insertion.

For this query a *readConcern=Majority* and a *ReadPreferences=SecondaryPreferred* was used. The majority read concern, instead of local, was choosen to be "almost"

sure that the campaigns shown, which could potentially receive donations, have been replicated and notified to the primary by most of the servers. Reading from the **majority snapshot** of the server may not show the latest campaigns, but certainly the most replicated ones.

TTL index

Without a proper solution, the following show the computational cost of ranking top donors using the Campaigns collection alone:

```
db.campaigns. explain("executionStats").aggregate(
 [
  {$unwind: "$donations"},
  {$match: {"donations.timestamp": {$gte: "2020-12-01",$lte: "2021-01-01"}}},
  {$group: {_id:"$donations.donor",totalDonated: {$sum:"$donations.amount"}}},
  {$sort:{totalDonated: -1}}
 ]
)
```

(a fictitious dates have been used for example)

```
{
      "queryPlanner" : {
            "plannerVersion" : 1,
             …
                  "inputStage" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
             …
      },
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 751,
            "executionTimeMillis" : 116,
            "totalKeysExamined" : 0,
            "totalDocsExamined" : 751,
            "executionStages" : {
             …
```

Since **there is no match at the beginning**, no document can be selected. Subsequent pipelines are done on all documents.

In addition, an unwind implies managing many more documents than those indicated in the execution plan. Indeed from the 751 documents about **108987 micro documents** (sigle donation) are obtained.

The system is subject to two types of queries that the campaign collection alone would not be able to perform. Furthermore, the ranking of the top monthly donors and categories implies limiting the search to only **monthly** donations transactions. Also, keeping track of those even beyond the month would make no sense (the rest are already in the campaign collection anyway). The first idea was to create a **Copped Collection** by truncating the space to a certain number of bytes that in the worst case should have been reserved. But the "worst trend" is a factor too much dynamic that would cause two types of events:

- the first is the *waste of space* when the monthly donations are few.
- the second is the *overwriting of the oldest donations*, but still monthly (and therefore of interest) if the donations are many.

To solve these two problems a **TTL index** was chosen.

A TTL index is defined on the values of the timestamp of *monthlyDonations* collection which reports, for each donation made in the campaigns collection, a single document related to the single donation behaving as if it were a log:

$db.monthlyDonations.createIndex(\ \{\ "timestamp": -1\ \},\{\ expireAfterSeconds: 2.592.000\}\ )$

With this index it becames a **montlhy** donations log.

The TTL thread will delete all documents whose timestamp is older than one month (the current one). As for the replicas, it could happen that in a direct reading on the replicas some data older than a month are also fetched as the TTL threads are not active on the secondary, but only on the primary. There will be a “eventual consistency” in which the replicas will still have to replicate the deletion of some transaction made by the primary thread on the server that was at that time elected as primary. However, these are minor problems in a social network and also this synchronization latency with the primary's oplog will be really short.

After the ttl index was applied, the documents to be treated for the aggregations decreased from 108968 to **9992**. Applying the same (much more simplified) aggregation to monthlyDonations’s collection, the following results are obtained:

```
db.monthlyDonations.aggregate
(
 [
  { $group: { _id: "$donor", totalDonated: { $sum: "$amount" } } },
  { $sort: {totalDonated: -1}}
 ]
)
```

```
{
      "queryPlanner" : {
            "plannerVersion" : 1,
            ...
                  "inputStage" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
            ...
      },
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 9992,
            "executionTimeMillis" : 41,
            "totalKeysExamined" : 0,
            "totalDocsExamined" : 9992,
            "executionStages" : {
            ...
```

It can be seen that time has decreased by a factor of 3. These results may not highlight the strong need for this new collection and a TTL index. This is because the campaign collection (and therefore all the donations) are all related to 2020. In 5 years, without this collection and TTL index, 5 years of documents should be scrolled and for each one make an unwind. It's obvious that it is infeasible.

Paragraph 10

Neo4J Design and Query Analysis

From the rest of the class diagram comes out that Neo4J will mainly deal with the management of Users, their Teams and their relationships. A small part of the management is also reserved for Campaigns which only need two fields to perform queries.

So, Neo4J will handle three types of nodes and six types of relationships.

Nodes


```
{
    “username”: “alanTurin”,
    “password”: abcdefg123456,
    “urlImage”: “http://...”,
    “region”: “Sicily”
    “province”: “Trapani”
    “cap”: 91011
}
```

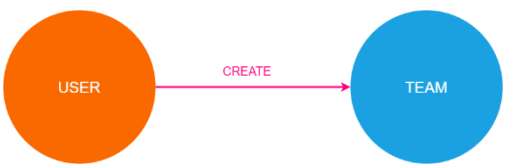
```
{
    “idTeam”: 173,
    “name”: “teamGiulia”
}
```

```
{
    “idCampaign”: “5b7d...12aa94”,
    “title”: “Aiutiamo la piccola Giulia ”,
    “urlImage”: “http://... ”
}
```

Relationships

The six relationships are:

- USER →CREATES→TEAM
  A user can create a team.

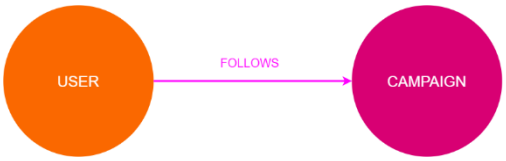


- USER →**BELONGS**→TEAM
  A user can belongs to a team.

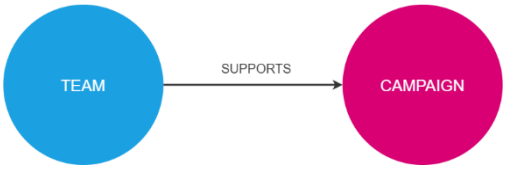
- USER →**IS_FRIEND_OF**→USER
  A user can be friend of other user.

- USER →**ORGANIZES**→CAMPAIGN
  A user can organize a campaign.

- USER →**FOLLOWS**→CAMPAIGN
  A user can follow a campaign.



- TEAM →**SUPPORTS**→CAMPAIGN
  A team can support a campaign.



Below, in Figure 5, a snapshot:

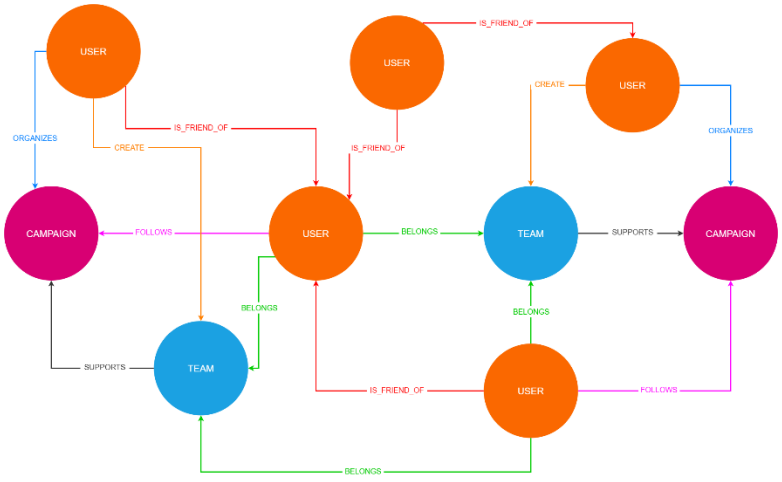


Figure 7

Let's see the main read and write queries handled by Neo4J.

Read operations		
Operation	Frequency	Cost
Retrieve all the informations about the user that is using the platform.	Medium	Low
Retrieve all the basic informations about campaings that the user created and followed .	Medium	Low
Retrieve the complete list of friends of user.	Medium	Low
Typical query For a specific user retrieve all the teams created and belongs to with their members and their supported campaigns.	Medium	Low/ Medium (it depends on the team)
Aggregation Retrieve N users based on distance, category followed and number of friends in common with a specific user.	Low	*Medium
Aggregation For a specific campaign retrieve the temporal trend of the average of followers.	Low	Low/ Medium (it depends on the campaign)

The field values with (*) are the cost that exploit the indexes (see after).

Write operations		
Operation	Frequency	Cost
Register a new user.	Medium	Low
Update url profile image of a specific user.	Low	Low
Add/Remove a user into/from the friends list.	Medium	Low
Add/Remove a member to/from the team.	Low	Low
Delete a team from own teams list of a specific user (if the user is an organizer, otherwise update the teams list it belongs to).	Low	Low
Create a new team.	Low	Low
Follow/Unfollow a campaign.	Hight	Low
Register a new campaign (preview).	Medium	Low

The only implementations for typical queries are shown below:

Typical queries implementations

For a specific user retrieve all the teams created and belongs to with their members and their supported campaigns.

```
MATCH (u:User)-[r:CREATES|BELONGS]->(t:Team) WHERE u.username=$username
WITH t as team
MATCH (u:User)-[r:CREATES|BELONGS]->(t:Team) WHERE ID(t)=ID(team)
WITH ID(t) as idTeam,t.name as nameOfTeam, u.username as username,
        CASE type(r) WHEN  "CREATES" THEN "organizer" ELSE "member" END AS rule
OPTIONAL MATCH (t:Team)-[s:SUPPORTS]-(c:Campaign) WHERE ID(t)=idTeam
WITH idTeam, nameOfTeam, username, rule, collect(DISTINCT ID(c)) as idCampaignsSupported
ORDER BY idTeam, rule DESC
RETURN idTeam, nameOfTeam as name, collect(username) as members, idCampaignsSupported
```

For a specific campaign retrieve the temporal trend of the average of followers.

```
MATCH (c:Campaign)-[f:FOLLOWS]-(u:User) WHERE c.idMongoDB=$idCampaign
RETURN f.timestamp as timestamp, count(*) as count
ORDER BY timestamp ASC
```

The typical query involving suggestions of friends will be analyzed in the next paragraph.

Paragraph 11

Neo4J Index and Constraint

All the queries managed by Neo4J are such as to exploit its native structure. Indeed, all queries are performing even in the absence of an index. The only query that strongly requires the presence of an index is that relating to the **suggestion of friends**. The friend suggestion policy tries to find more compatible users around a specific user according to the physical distance and the number of categories followed and relating to campaign created that they have in common. The ranking of candidates is thus drawn up by taking the first 3 and recommending them to the user.

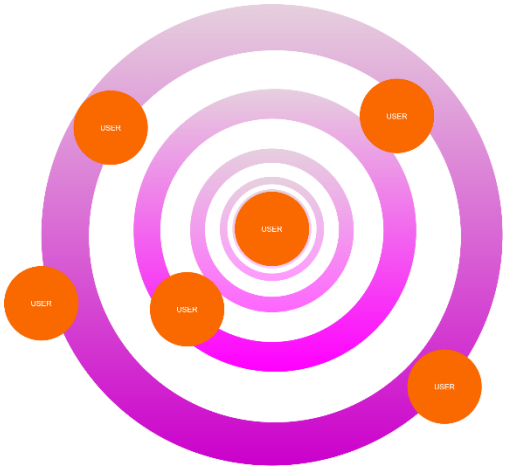


Figure 8: proximity radar.

Without the use of an index that keeps users sorted by physical proximity, running the query would be expensive even using algorithms such as merge sort or quick sort because even if $O(nlogn)$, where the number n is the total number of users, is a good complexity, in a social network n is really high.

During registration, the user is asked to also enter the **postcode** of the city in which he lives. In Italy the postal codes are organized in such a way as to be minimal starting from Sardegna, and then from Northern Italy to increase more and more down to Sicily. Doing so, by creating the following *single-property index*:

```
CREATE INDEX proximityOfUsers FOR (n:User) ON (n.cap)
```

it can achieve a quick search.

Before proceeding, the query that picks up the top 3 suggested friends is given below:

On the code side, there is a need to define two operators.

```
String operator1 = "WHERE NOT (n)-[:FRIEND_OF]-(u) AND u.username=$username AND n.cap<=u.cap AND
                    n.username<>$username"
String operator2 = "WHERE NOT (n)-[:FRIEND_OF]-(u) AND u.username=$username AND n.cap>=u.cap AND
                   n.username<>$username";
boolean decision = new Random().nextBoolean();
```

Depending on the random result of *decision* will be used *operator1* (≤) as operator (therefore the search for suggested friends will be done considering all candidates with a cap less than or equal to that of the user), otherwise the *operator2* (≥) will be used (hence the search for friends will be made considering all candidates with a cap greater than or equal to that of the user). The query is the following:

find all the campaigns followed or created by the current user and create a list with the related categories, then find the first 15 users who are not yet friends of him and sort them by increasing cap (decreasing)

```
MATCH (u:User {username:$username})-[r]-(c:Campaign)
WITH collect(DISTINCT c.category) as favoriteCategoryOfCurrentUser
MATCH (n:User),(u:User) ((decision==true)?(operator1):(operator2))
WITH n.username as candidateFriend, n.cap as candidateCap, favoriteCategoryOfCurrentUser, rand() as
      randomValue
((decision==true)?(ORDER BY candidateCap DESC):(ORDER BY candidateCap ASC)) +
LIMIT 15
```

```
MATCH (u:User)-[r]-(c:Campaign) WHERE u.username = candidateFriend
WITH u, collect(c.category) as favoriteCategoryOfCandidate, favoriteCategoryOfCurrentUser, size([y IN
      collect(DISTINCT c.category) WHERE y IN favoriteCategoryOfCurrentUser]) as inCommon, randomValue
RETURN u.username as username, u.urlProfileImage as urlProfileImage, u.region as region, u.province as
          province, u.cap as cap, inCommon, randomValue
ORDER BY randomValue DESC, inCommon DESC
LIMIT 3
```

for each candidate, finds his favorite categories based on the campaigns he follows or has created and finds how many he has in common with the current user. Assign a random number and take those with a higher random number and in case of a tie, choose those with higher categories in common.

The two operators defined initially are necessary in order not to take only candidates with greater or only lesser cap. In fact, it could happen that a user has the largest cap code, if we ordered by higher cap codes he would never be suggested any friend. The

assignment of a random number is done to avoid that the 3 suggested candidates are always the same.

The choice to initially limit to only 20 candidates is due to the computational cost that the query has in the subsequent stages. It will be understandable in the analysis that will be made below.

Below is the **PROFILE** of query **without the use of any index**:

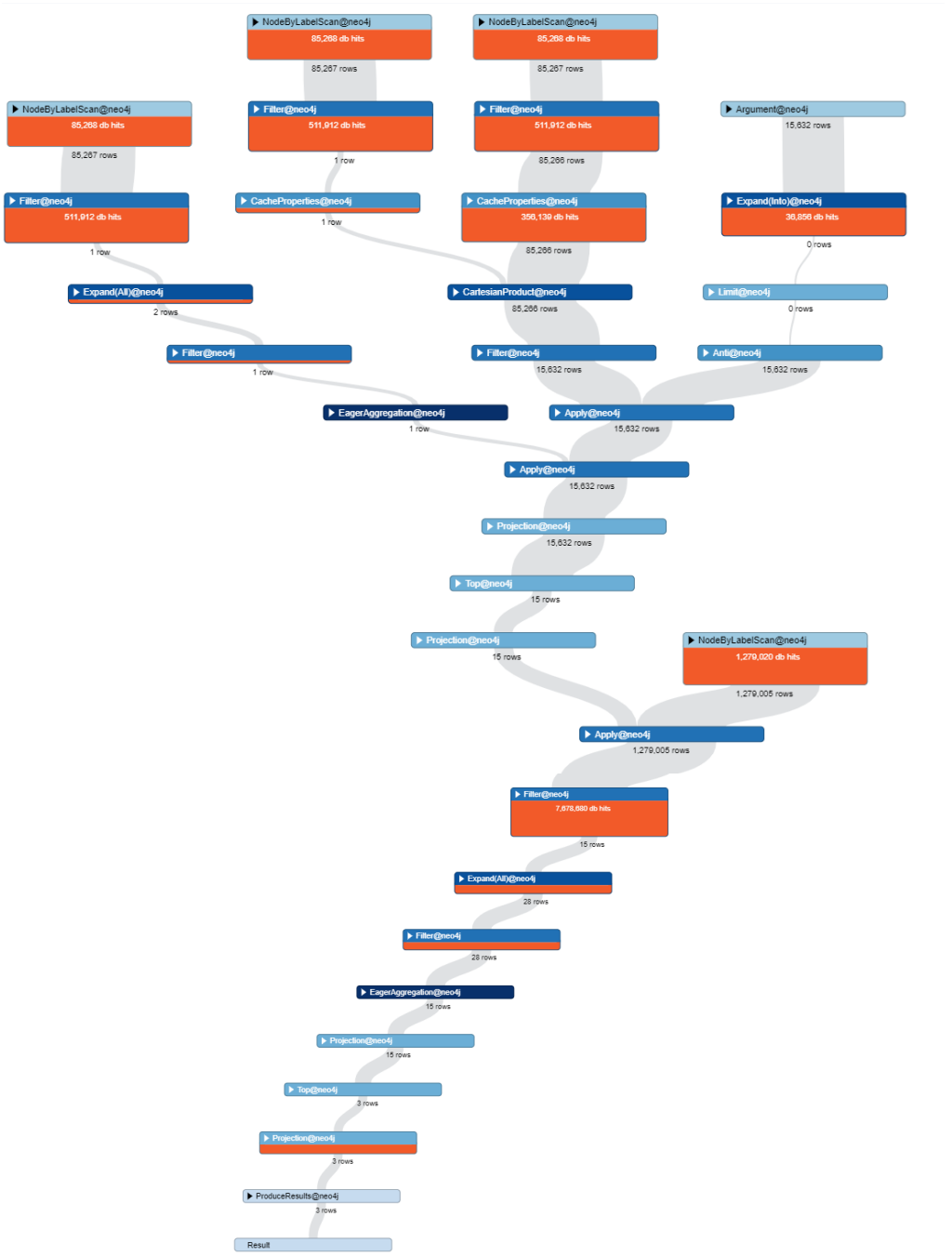


Figure 9: profile of query without any index.

The two important aspects to which attention must be paid are the **number of results that each stage gives as output** and above all the **db hits** in ORANGE. Each operator in a query will ask the Neo4j storage engine to do a particular work. A *database hit,* from the documentation, "*is an abstract unit of this storage engine work*". As we can see the query is asking to do too much work, especially due to the fact that no data structure such as indexes can support it. The number 15 of candidates for which to search for the respective preferred categories already is necessary to lighten the load. But what weighs the most is the sorting by cap. There is no data structure that holds this order, so Neo4J accesses all nodes. Before seeing the computational cost with the index, let's see what is the cost after the introduction of a particular **constraint**, the one on the *username*, which must be **unique**:

```
CREATE CONSTRAINT uniqueUsername
ON (user:User) ASSERT user.username IS UNIQUE
```

to respect the constraint Neo4J will automatically insert a BTREE type index.

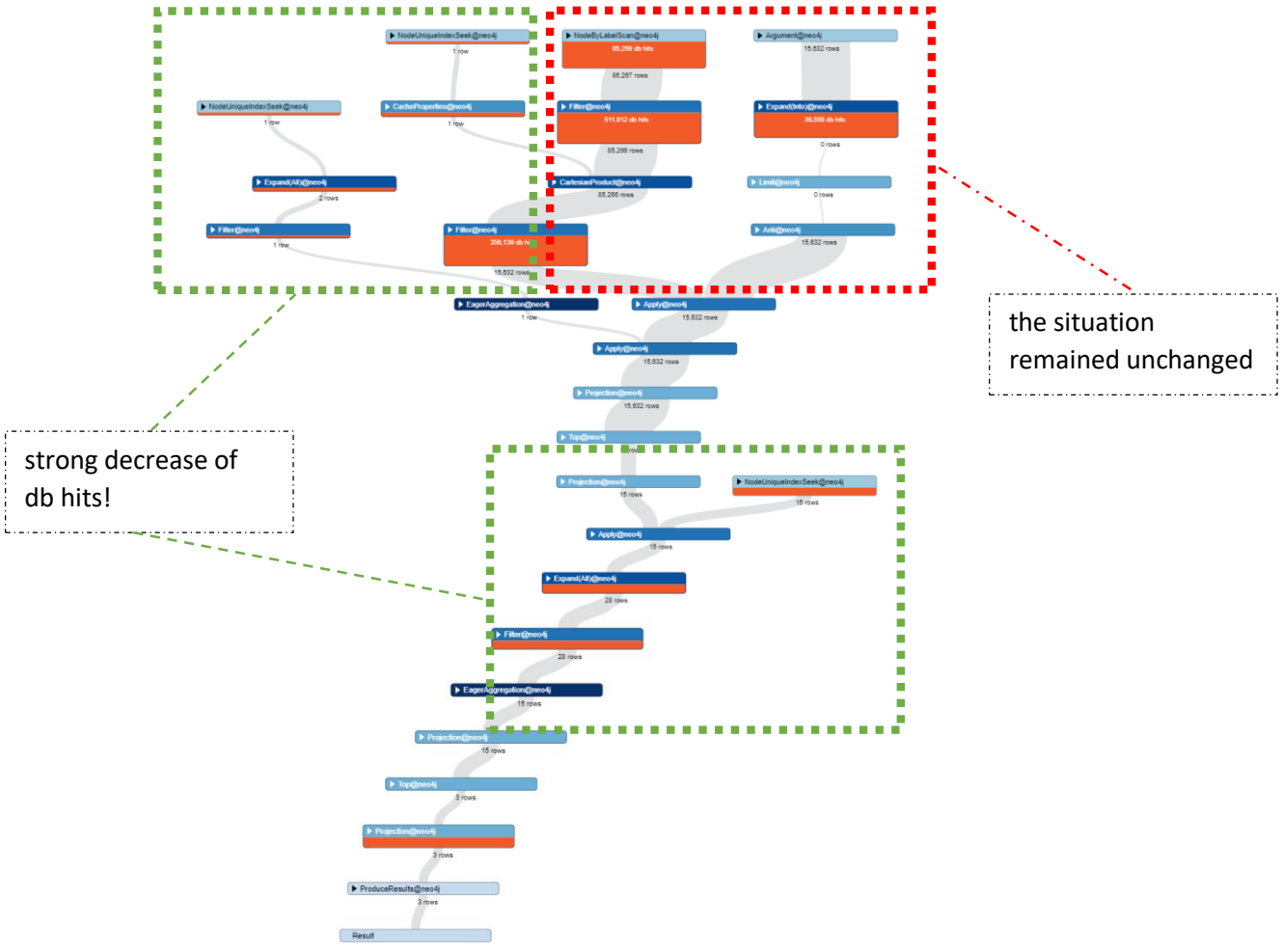


Figure 10: profile query with constraint on username.

The ideal would now be to reduce the part framed in red.

Below is the computational cost **using the *proximityUsers* index above defined**:

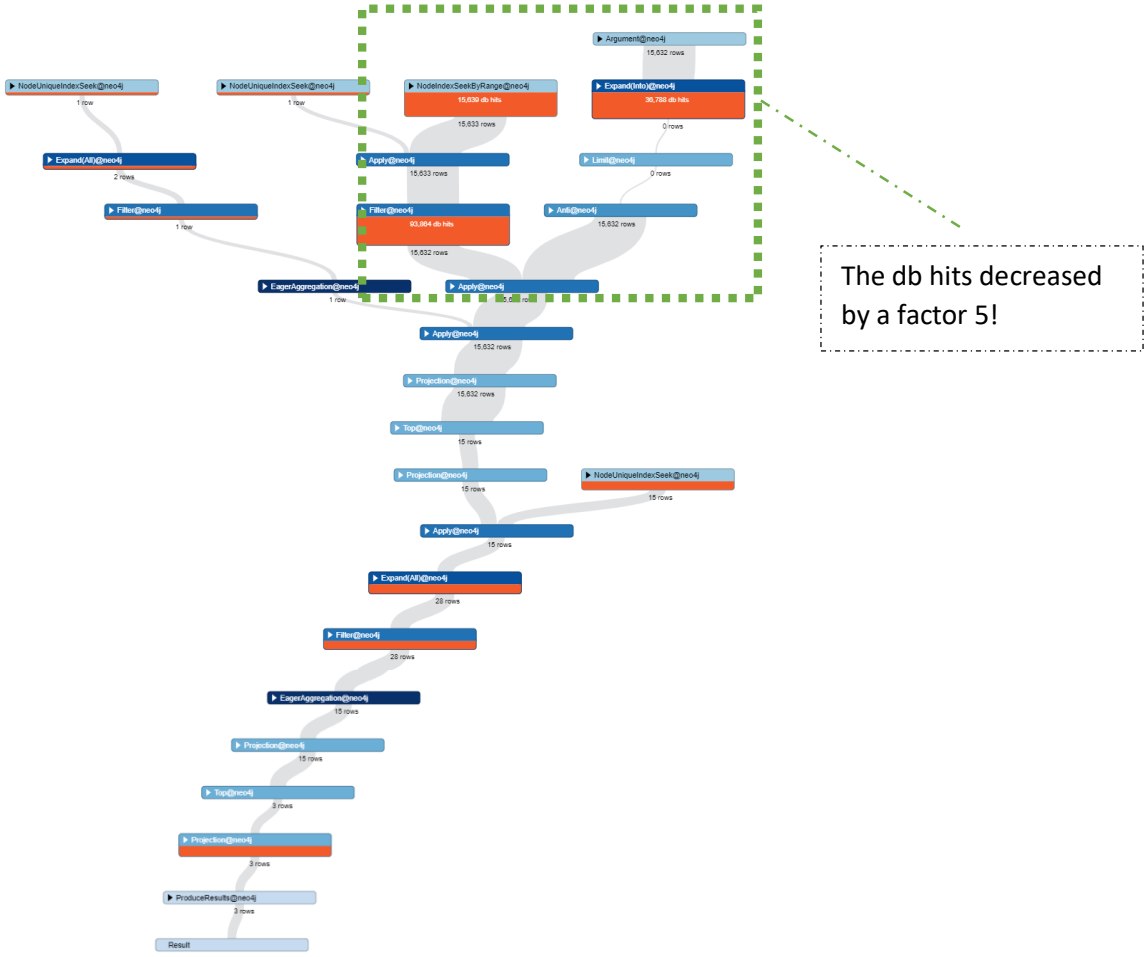


Figure 11: profile query with constraint on username and index on cap.

The use of the two indexes greatly decreases the db hits, therefore the storage workload in recovering the data. In a real social network, the storage size is thousands of times larger than this. Such a drastic decrease in db hits is vital.

Paragraph 12

The Individual Consistency, Collection-Consistency and the Cross-Consistency solution

Individually the two dbms guarantee different types of consistency.

With the **MongoDB** master slave model the writes will be done directly on the primary thus ensuring **monotonic writes**. Using a *writeConcern* of type *W3* for the writings (which regards only the "creation of the campaign" and "donation to a campaign") it is ensured that the user can have control back only after all the writings have been replicated and all replicas have notified the primary. Other users, however, may see an inconsistent state of the data as they can read from secondary. So the consistency model is of the type ***read your write consistency***.

Neo4j, being a standalone server, ensures **strong consistency**.

The real problem is when the data goes from a MongoDB to Neo4J. The only data that are subject to this series are campaigns. A part of them, such as MongoDB's ObjectId, the title and the url of the image are also copied to Neo4J.

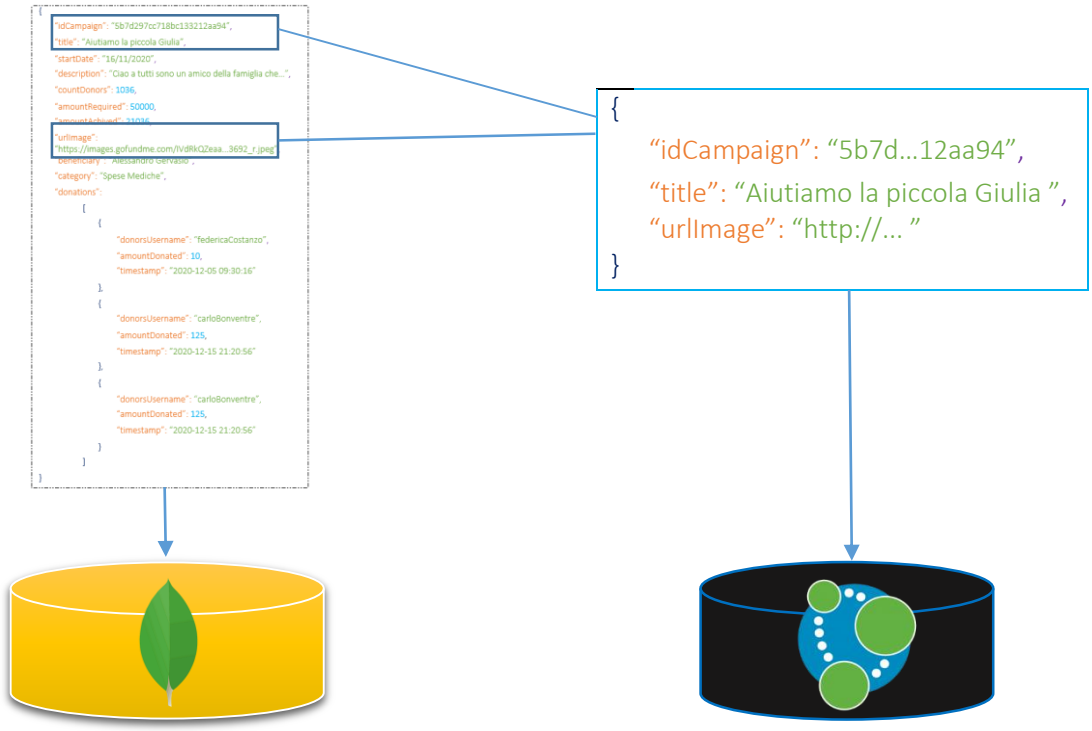


Figure 12: splitting of campaign document.

The need to have this data on Neo4J will be clear in paragraph X.

Only one problem can happen: the campaign is registered in MongoDB, but not in Neo4J.

MongoDB ensures **document-level atomicity**. Changing several documents even from the same collection does not ensure roolback in the event of failure. **Collection-level atomicity** is needed when we have new scraping-generated campaigns on GoFoundMe. They already have a list of donations. It is necessary to atomically register for each campaign in the campaign collection all the relative monthly donations in the monthlyDonations collection.

It is necessary to use the transactions, defined by the following pseudo java code:

```
     //open a session
     final ClientSession clientSession =
              Connection.getInstance().getMongoClient().startSession();


    //define body transaction...
    TransactionBody transaction = new TransactionBody<Void>() {


   @Override
   public Void execute() {


        1)insert the campaign into campaign's collection

        2)for each related donation, insert it into montlyDonation's collection




       return null;

   }
};

    //start transaction...
    try {
           clientSession.withTransaction(transaction);
    }catch (Exception e) {
           System.out.println("an error occurs during

                    transaction..."+e.getLocalizedMessage());
    }
```

In this way we are however not guaranteed **cross-consistency**. The idea is that MongoDB roolbacks when an exception within the execute method is raised. Then the method that takes care of transferring the campaign to Neo4J is moved within

the transaction, even if it is not a MongoDB operation. In this way, if an exception is thrown or any other type of error occurs (in this case we explicitly the exception), the total rollback is performed:

```
     //open a session
     final ClientSession clientSession =
               Connection.getInstance().getMongoClient().startSession();


    //define body transaction...
    TransactionBody transaction = new TransactionBody<Void>() {


   @Override
   public Void execute() {


        1)insert the campaign into campaign's collection

        2)for each related donation, insert it into montlyDonation's collection

        3)insert a (preview) campaign into Neo4J.
          3.1) if it fails but raises no exceptions
                  throws explicitly an exception



       return null;

   }
};

    //start transaction...
    try {
           clientSession.withTransaction(transaction);
    }catch (Exception e) {
           System.out.println("an error occurs during

                    transaction..."+e.getLocalizedMessage());
    }
```

Paragraph 13

The Exploration Mode

What happens if MongoDB (all replicas) fail?

To understand how the platform behaves it is necessary to anticipate the graphic interface of InstaFound:

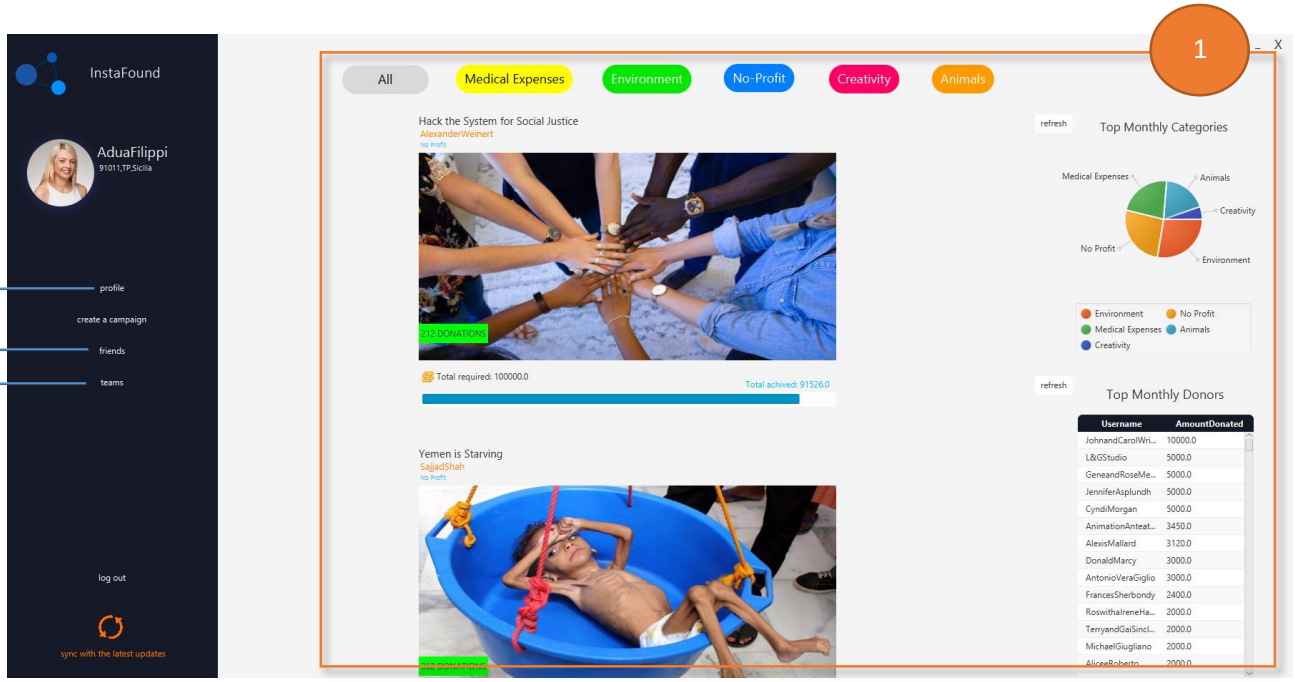


Figure 13: home screen of Instafound.

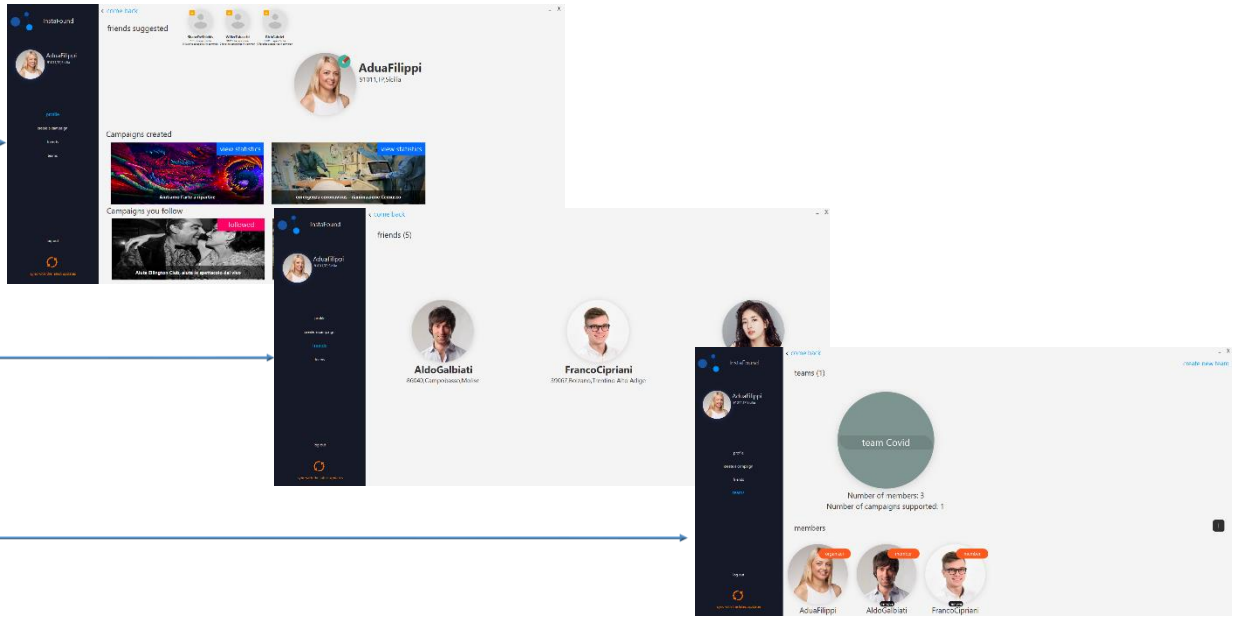


Figure 14: profile, friends list and teams screen of Instafound.

1 MongoDB is responsible for maintaining the data that will be presented in the **Home**:

- list of the most recent campaigns that can be filtered by category.
- monthly statistics of the categories.
- ranking of the top donors of the month

2 Neo4J is responsible for all data that appears in the **profile**:

- suggestion of friend.
- preview of the created campaigns.
- preview of the followed campaigns.
- user information.
- Login/logout

it is also responsible for the **friends** and **team** (including members) **lists**.

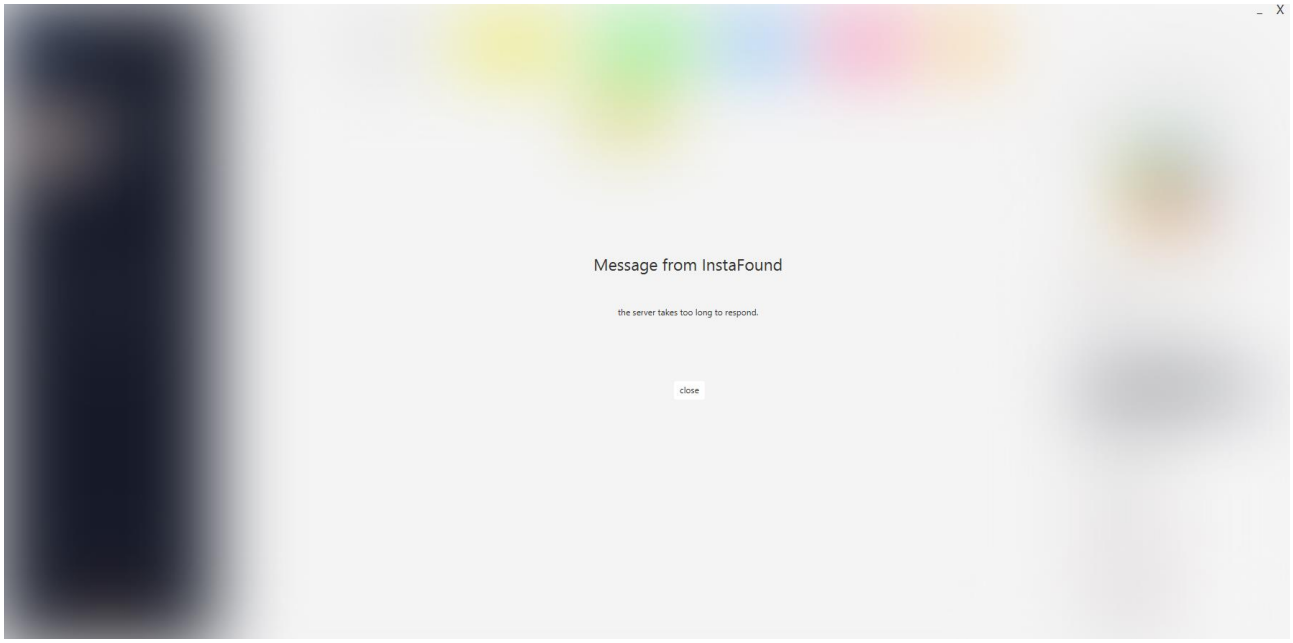


Figure 15: message about the MongoDB servers not working

So if MongoDB fails the user can still continue browsing around everything related to him such as the profile, the list of friends and teams (members included). Obviously all related actions such as follow / unfollow a campaign, remove a friend, create / delete / leave a team, add / remove a member are still possible. Even login / logout, as managed entirely by Neo4J, continue to be possible.

Now it is even clearer **why it is necessary to have that fragment of the campaign document shared between MongoDB and Neo4J**, to show the previews of the campaigns even in these circumstances.

What happens if Neo4J fails?

Many of the features would be impossible to use. Instead of closing everything, the so-called *ExplorationMode* has been designed in which the user can only browse the home and see the latest campaigns, the monthly statistics of the categories and donors. In short, everything that is managed by MongoDB. This is possible thanks to this clear separation of data and roles.

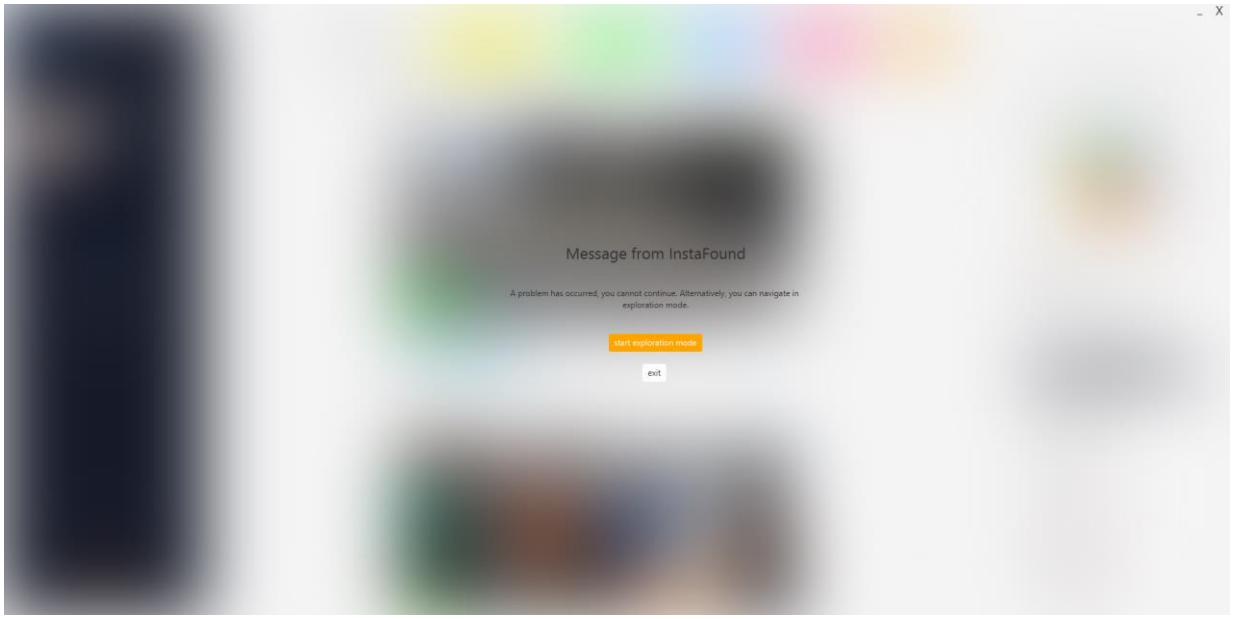


Figure 16: message about the Neo4J server not working

As you can see from figure 10, the user can log out (exit) or start the *Exploration Mode*:

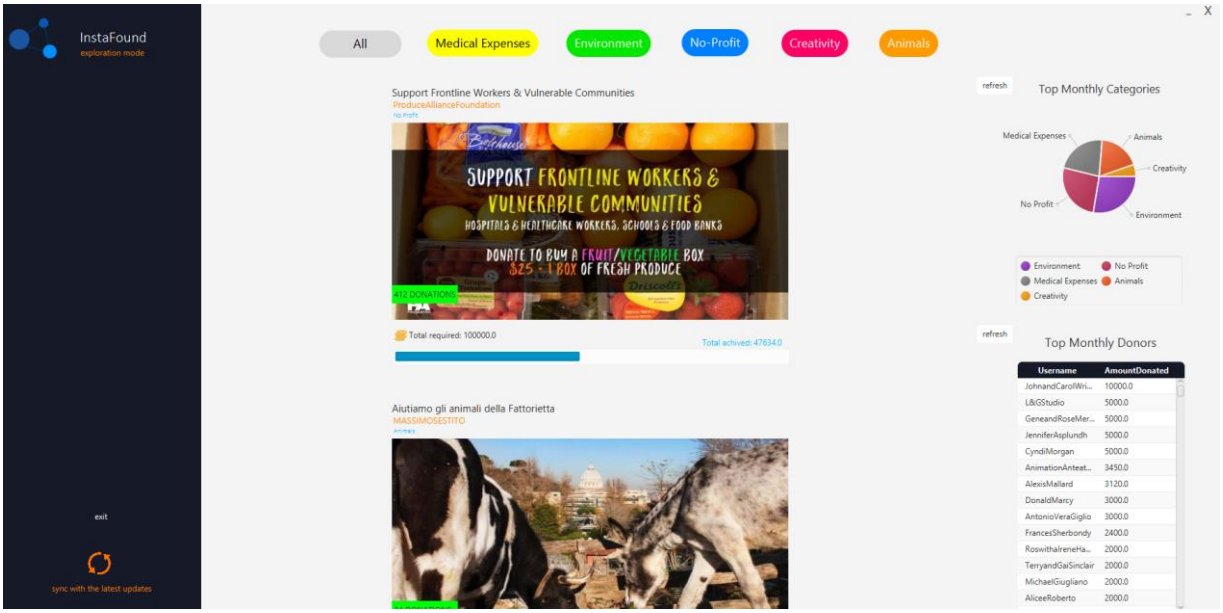
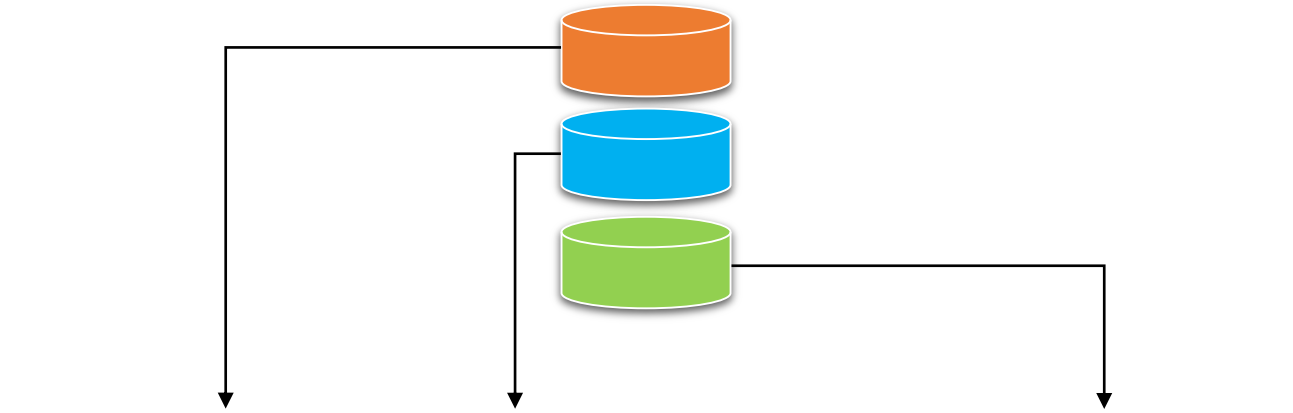


Figure 17: the user interface of *Exploration Mode*.

Paragraph 14
Sharding

Like all social networks, servers may have to manage large datasets and an high iterations by users. With the **master slave** model the primary may be subjected to too many iterations since it is the only one to manage the writes. The *vertical scalability* solution is too expensive and limiting. *Horizontal scaling*, via **sharding**, is the right solution in this case.

Having 3 virtual machines available, we could forecast the presence of 3 shards, each with two replicas. Each shard would be placed on a virtual machine different from the other 2 shards, while the replicas on the remaining different machines.



172.16.3.116	172.16.3.117	172.16.3.118
Replica shard1 (mongod)	Replica shard2 (mongod)	Replica shard3 (mongod)
Replica shard2 (mongod)	Replica shard1 (mongod)	Replica shard2 (mongod)
Replica shard3 (mongod)	Replica shard3 (mongod)	Replica shard1 (mongod)
Neo4J instance		
Router (mongos)	Router (mongos)	Router (mongos)
Replica Config Server (mongod)	Replica Config Server (mongod)	Replica Config Server (mongod)

In addition to the individual shards, we have provided 3 **config servers** to maintain the metadata of each shard, also with replicas to increase availability, and finally 3 **mongos routers** for client / server interfacing to prevent a single instance from acting as a bottleneck, even if they are efficient enough.

We exclude **hashed sharding** for the shard key because most queries involve aggregation of campaigns belonging to the “same family". For example, fetching all the campaigns of a specific category and choosing to hash the *ObjectId* would not be efficient. In fact, we should access different shards to retrieve all those campaigns since the hash function on ObjectId distributes the data in a homogeneous way but certainly scattering the campaigns too much among the servers.

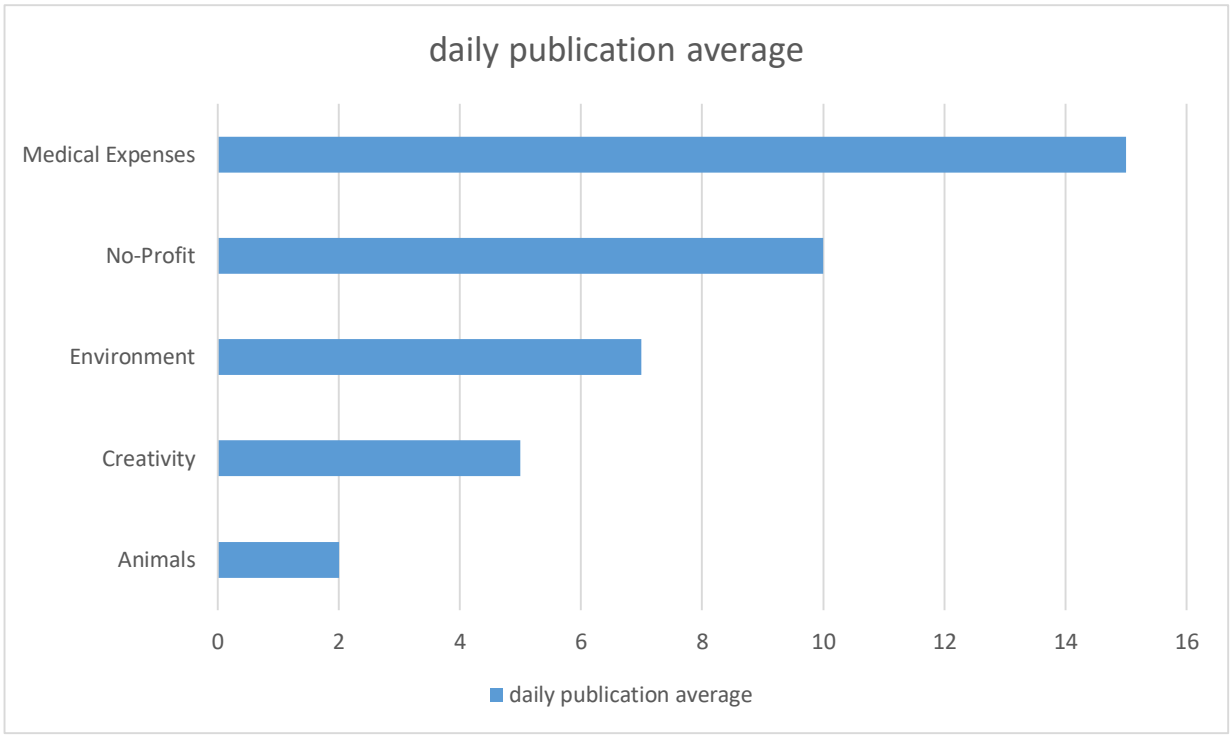
About the **range-based sharding** a candidate could be the *startDate*, leaving automatic the background processes of splitting and balancing. The date field have the form of “yyyy/mm/dd”, and if mongos create at most one chunk per day the problem will be when the platform will become viral, because may will be too many campaign creation dates in place in the same day. But the splitting process will not be able to split when the chunk will exceed the max chunk size decided in config.settings because every document has the same value for the shard key. These **jumbo chunks** are to be avoided.

The best shard key is made in the **category field**. However, we must disable the automatic splitting and balancing processes and choose to *tag* the individual servers and associate them with customized ranges of values that each one will have to deal with. This is because the single category values have different distributions and because it is better not to scatter the same values on different servers given the nature of the aggregations and queries.

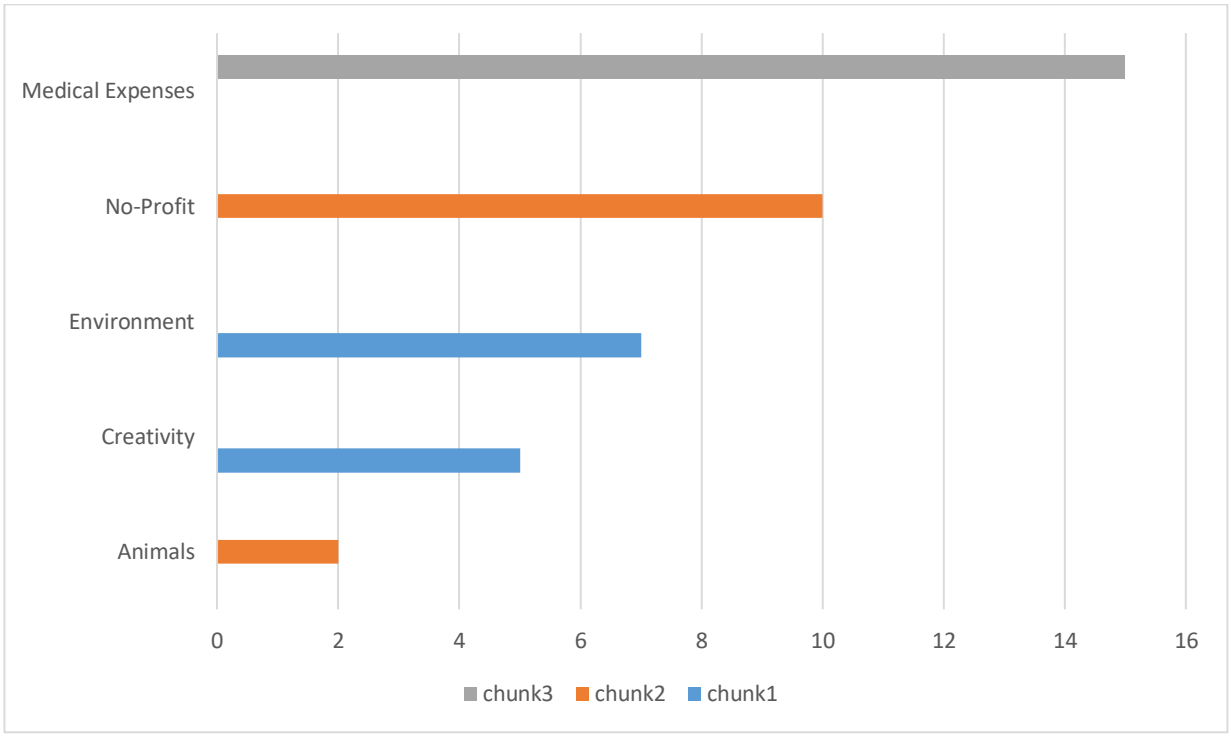
To predict the distribution of values we have made an aggregation on the data in possession until today:

```
db.campaigns.aggregate
(
 [
  {$group:{_id: {"startDate":"$startDate", "category": "$category" }, totDaily: {$sum: 1}}},
  {$group: {_id: "$_id.category", avgDaily: {$avg: "$totDaily"}}}
 ]
)
```

The result shows an average trend of publications shown below:



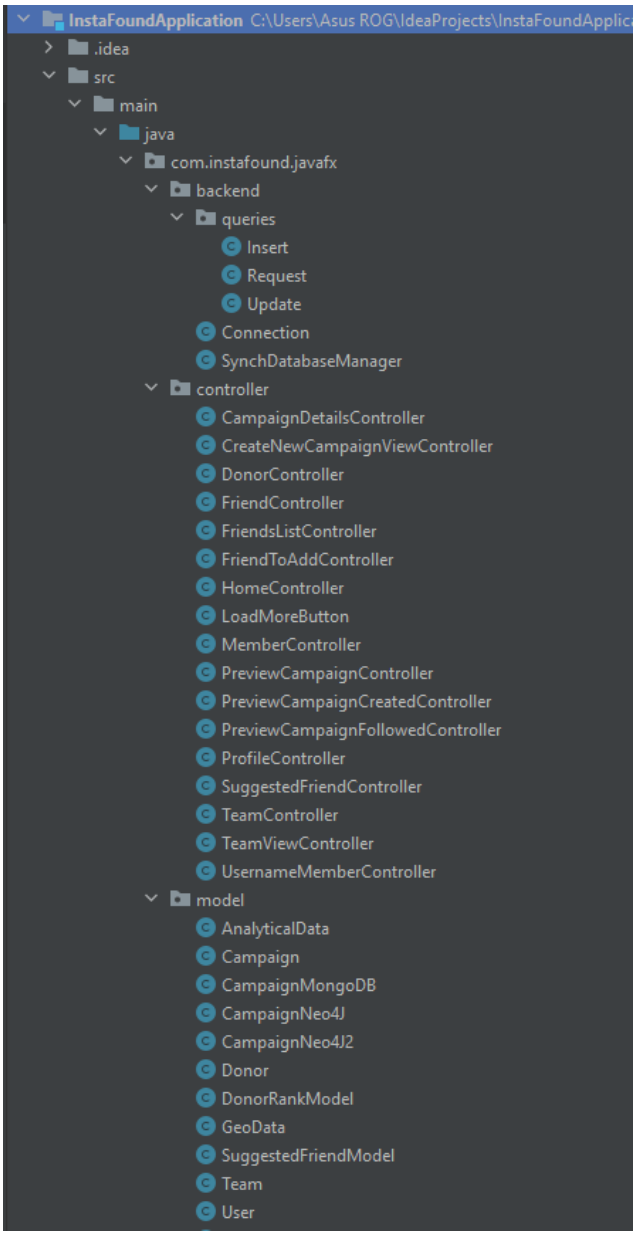
Having only 3 machines available, the range that each shard will have to deal with, in order to distribute the load evenly, will be the following:

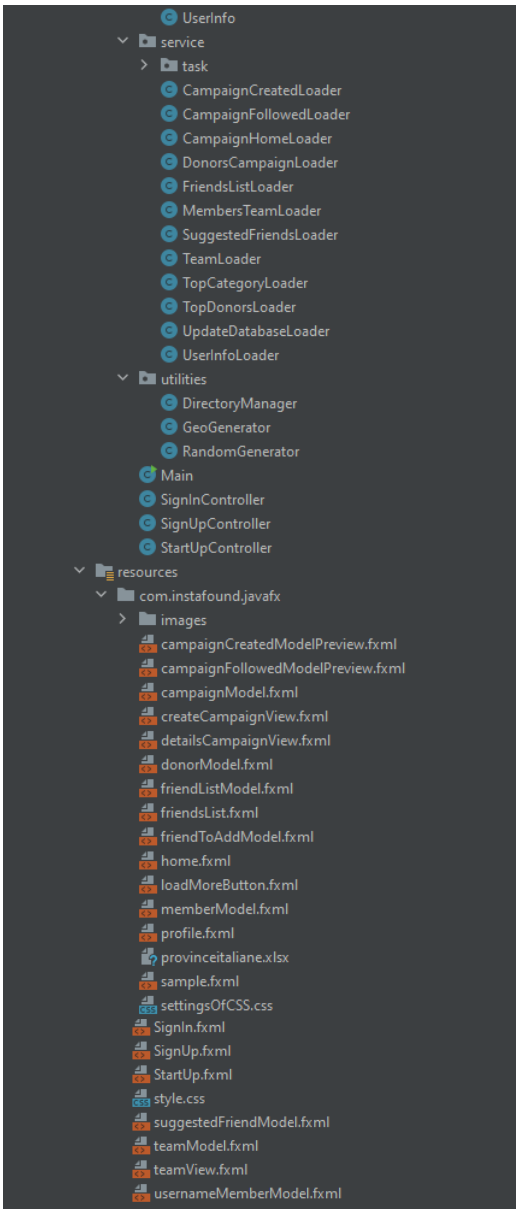


Paragraph 14

Main concepts of implementation

InstaFound is a sofware programmed using *Java 8* as programming language combined with *JavaFX* in Intellij IDE using *Maven* to manage easily the dependencies. The whole project consists of more than 12.500 row of codes (fxml not included) and it's organized in the following package:





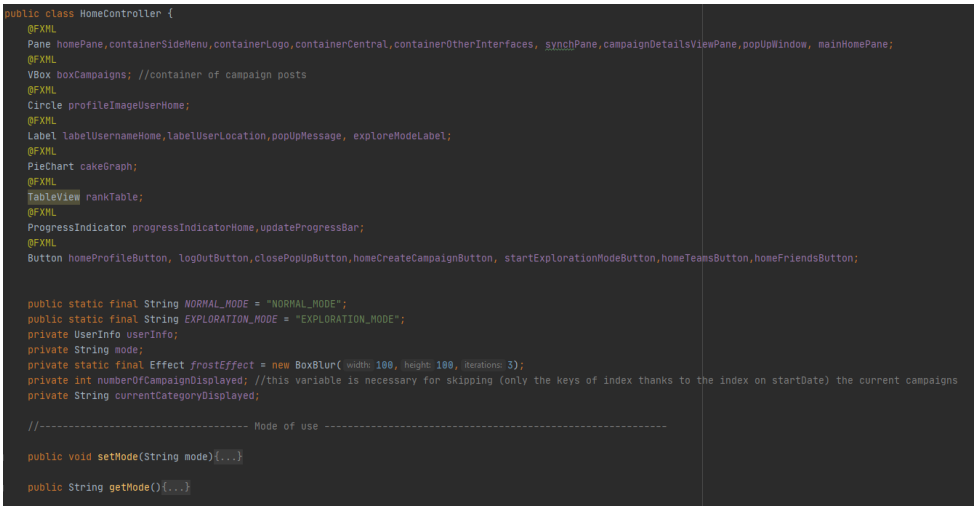
As you can see we used the *Model View Controller* paradigm. The fxml code was automated by *Scene Builder*. Obviously on the backend side MongoDB and Neo4J were used with their respective java drivers. For scraping, *jsoup* was used to extract

static content, the payload of the *http responses* was also manipulated in order to have more precise data, and finally *Selenium* (with *chrome web drive*r) was used for dynamic content. The following are de dependecies managed by Maven:

```
<dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-sync</artifactId>
        <version>4.1.1</version>
    </dependency>
    <dependency>
        <groupId>org.neo4j.driver</groupId>
        <artifactId>neo4j-java-driver</artifactId>
        <version>4.2.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.15</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>3.15</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.6</version>
    </dependency>

</dependencies>
```

We avoid reporting the structure of each class for reasons of space. Below are only the **main controllers** that control the **5 sections** in which the platform is organized (see the next paragraph for more details on the UI):

- HomeController.java
  It is in charge of controlling all the other 4 controllers. and to show the latest campaigns filtered by category, the monthly top categories and the ranking of the top donors.



```
public class HomeController {
    @FXML
    Pane homePane,containerSideMenu,containerLogo,containerCentral,containerOtherInterfaces, synchPane,campaignDetailsViewPane,popUpWindow, mainHomePane;
    @FXML
    VBox boxCampaigns; //container of campaign posts
    @FXML
    Circle profileImageUserHome;
    @FXML
    Label labelUsernameHome,labelUserLocation,popUpMessage, exploreModeLabel;
    @FXML
    PieChart cakeGraph;
    @FXML
    TableView rankTable;
    @FXML
    ProgressIndicator progressIndicatorHome,updateProgressBar;
    @FXML
    Button homeProfileButton, logOutButton,closePopUpButton,homeCreateCampaignButton, startExplorationModeButton,homeTeamsButton,homeFriendsButton;


    public static final String NORMAL_MODE = "NORMAL_MODE";
    public static final String EXPLORATION_MODE = "EXPLORATION_MODE";
    private UserInfo userInfo;
    private String mode;
    private static final Effect frostEffect = new BoxBlur( width: 100, height: 100, iterations: 3);
    private int numberOfCampaignDisplayed; //this variable is necessary for skipping (only the keys of index thanks to the index on startDate) the current campaigns
    private String currentCategoryDisplayed;

    //------------------------------------ Mode of use ------------------------------------------------------------

    public void setMode(String mode){...}

    public String getMode(){...}
```

```
//is set by task UserInfoLoaderTask
public void setUserInfo(UserInfo userInfo){...}

public void updateUserUI() {...}

private void initCakeGraph(){...}

public void updateCakeUI(ArrayList<AnalyticalData> topCategories){...}

private void initRankingOfDonors() {...}

public void updateDonorsRankingUI(ArrayList<DonorRankModel> topDonors){...}

//load last campaigns
private void loadLastCampaigns(){...}

public void loadHome(){...}


//---------------------------------- SECTIONS OF PLATFORM ------------------------------------------------------

public void openUserProfile(MouseEvent mouseEvent) {...}

public void openFriendsList(MouseEvent mouseEvent) {...}

public void openTeamView(MouseEvent mouseEvent) {...}

public void openCreateNewCampaignView(MouseEvent mouseEvent) {...}

//open details of preview campaign (preview is intedend as preview campaign of neo4j, so we need to retrieve all the info)
public void openCampaignDetailsView(ObjectId idCampaign){...}

//open details of preview campaign (preview is intedend as preview campaign of mongodb, so we have just all we need)
public void openCampaignDetailsView(CampaignMongoDB campaignMongoDB){...}

public Pane getContainerCentralHome() { return containerCentral; }
```

```
 public Pane getContainerCentralHome() { return containerCentral; }

 public Pane getContainerOtherInterfaces() { return containerOtherInterfaces; }

 //given i highlights the i° button of the home interface
 public void highlightsButton(int i){...}

 //---------------------------------- STATSYSTICS ------------------------------------------------------------------

 public void refreshCakeUI(MouseEvent mouseEvent) {...}

 public void refreshDonorsRanking(MouseEvent mouseEvent) {...}

 public void addCampaignToHomeUI(Parent root) { boxCampaigns.getChildren().add(root); }

 public void removeCampaignsFromHomeUI(){...}

 //----------------------- Loading Category Buttons-----------------------------------------------------

 public void loadMedicalExpensesCategory(MouseEvent mouseEvent) {...}

 public void loadEnvironmentCategory(MouseEvent mouseEvent) {...}

 public void loadNoProfitCategory(MouseEvent mouseEvent) {...}

 public void loadCreativityCategory(MouseEvent mouseEvent) {...}

 public void loadAnimalsCategory(MouseEvent mouseEvent) {...}

 public void loadAllCategory(MouseEvent mouseEvent) {...}

 public String getCurrentCategoryDisplayed() { return currentCategoryDisplayed; }

 public void removeLoadMoreButton(){...}
```

```
//-------------------------- Frost Home Pane ----------------------------------------------
public void frostHomePane(){...}

public void unFrostHomePane(){...}

//-------------------------- POPUP WINDOW ------------------------------------------------

public void showPopUpMessage(String mex){...}

public void closePopUpMessage(MouseEvent mouseEvent) {...}


//-------------------------- Log Out and Close/Reduce window -----------------------------------------------------

public void logOut(MouseEvent mouseEvent) {...}

public void closeMainWindow(MouseEvent mouseEvent) {...}

public void reduceMainWindow(MouseEvent mouseEvent) {...}

//-------------------------------------------- OTHER METHODS-------------------------------------------------

//this method restart the social into a exploration mode (it's called only when the are neo4j's problem)

public void showPopUpOfExplorationMode(){...}

public void restartAsExplorationMode(MouseEvent mouseEvent) {...}

public void setVisibilityProgressIndicator(boolean visibility){...}

public int getNumberOfCampaignDisplayed() { return numberOfCampaignDisplayed; }

public void incrementNumberOfCampaignDisplayed() { numberOfCampaignDisplayed += 1; }

public void destroyPane(){...}

  //-------------------------------------- SYNCH DATABASE -------------------------------

  public void synchDB(MouseEvent mouseEvent) {...}

  public void showUpdateWindow(){...}

  public void hideUpdateWindow(){...}

  public void setUpdateProgressBar(double val){...}
```

- ProfileController.java
  It is in charge of managing the previews of the campaigns created and followed by the user and suggesting new friends.

```
public class ProfileController implements Initializable {

    @FXML
    Label labelUsernameProfile;
    @FXML
    Pane mainProfilePane,statisticsPane,containerCentralHome,containerProfile,newUrlPane;
    @FXML
    private Circle circleImageView;
    @FXML
    private HBox containerCampaignsCreated, containerCampaignsFollowed,containerSuggestedFriends;
    @FXML
    private ProgressIndicator progressIndicatorCampaignsCreated,progressIndicatorCampaignsFollowed,progressIndicatorFriendsSuggested;
    @FXML
    BarChart barChartStatisticsCampaign;
    @FXML
    Button showDonationsTrendButton, showFollowersTrendButton;
    @FXML
    Label labelLocationProfile,titleStatisticLabel;
    @FXML
    NumberAxis barCharYLabel;
    @FXML
    TextField textFieldNewUrl;

    UserInfo userInfo;
    private HomeController homeController;

    private ArrayList<PreviewCampaignCreatedController> campaignCreatedControllers;
    private ArrayList<PreviewCampaignFollowedController> campaignFollowedControllers;
    private PreviewCampaignCreatedController selectedCampaignCreated;

    private static final Effect frostEffect = new BoxBlur( width: 100, height: 100, iterations: 3);
    private static SimpleDateFormat sdf = new SimpleDateFormat( pattern: "yyy-MM-dd");
    private static Calendar cal = Calendar.getInstance(TimeZone.getTimeZone("Europe/Italy"));
```

```
    private void loadUserProfileImage() throws NullPointerException{...}

    public void loadSuggestedFriends(){...}

    public void setHomePanel(HomeController homeController, Pane containerCentralHome, Pane containerProfile){...}

    public String getUsernameOfCurrentUser(){...}

    public ProgressIndicator getProgressIndicatorFriendsSuggested(){...}

    public HBox getContainerSuggestedFriends(){...}

    public void initProfile(UserInfo userInfo){...}

    @Override
    public void initialize(URL location, ResourceBundle resources) {...}

    public void addCampaignCreated(PreviewCampaignCreatedController previewCampaignController){...}

    public void clearCampaignCreated(){...}

    private void frostMainPane(){...}

    private void unFrostMainPane(){...}

    //------------------------------------------------ STATISTICS CAMPAIGN ----------------------------------------------------------------------

    public void openStaticsPane(PreviewCampaignCreatedController previewCampaignCreatedController){...}

    public void closeStatisticsPane(){...}

    public void backToHome(MouseEvent mouseEvent) {...}

    public void showDonationsTrend(MouseEvent mouseEvent) {...}

    public void showFollowersTrend(MouseEvent mouseEvent) {...}

    private void loadBarChart(String title,
                              String yLabel,
                              ArrayList<AnalyticalData> averageOfData,
                              SimpleDateFormat parser){...}

    //------------------------------------------------ DETAILS CAMPAIGN ------------------------------------------------

    public void openDetailsCampaign(ObjectId idCampaign){...}

    public void removeFollowedCampaignFromUI(PreviewCampaignFollowedController previewCampaignFollowedController) {...}

    public void addFollowedCampaign(PreviewCampaignFollowedController campaignFollowedController) {...}

    public void closeUrlPane(MouseEvent mouseEvent) {...}

    public void openUrlPanel(MouseEvent mouseEvent) {...}

    public void saveNewUrl(MouseEvent mouseEvent) {...}

    public void showPopUpMessage(String s) {...}

    //---------------------------------------- EXPLORATION MODE ---------------------------------

    public void showPopUpOfExplorationMode() {...}

    public void destroy() {...}

}
```

- CreateNewCampaignController.java
  It is in charge of managing the creation of new campaigns.

```
public class CreateNewCampaignController implements Initializable {

    @FXML
    Pane containerCentralHome,containerOtherInterfaces;
    @FXML
    Button medicalExpensesButton,environmentButton, noProfitButton,creativityButton,animalsButton;
    @FXML
    TextField titleNewCampaign,urlImageNewCampaign,amountRequiredNewCampaign,beneficiaryNewCampaign;
    @FXML
    TextArea descriptionNewCampaign;
    @FXML
    CheckBox teamNewCampaignOption;
    @FXML
    ListView listViewOfTeams;
    @FXML
    ProgressIndicator progressIndicatorListOfTeam;
    @FXML
    Label mexCreationCampaign, categoryLabel;

    private ArrayList<Team> teams;
    private HomeController homeController;

    private String categorySelected;
    private UserInfo userInfo;
    Button selectedCategoryButton;
    private SimpleDateFormat formatTimestamp = new SimpleDateFormat( pattern: "yyyy-MM-dd");


    public void setHomePanel(Pane containerCentralHome, Pane containerOtherInterfaces){...}

    public void setUserInfo(UserInfo userInfo) { this.userInfo = userInfo; }

    public void backToHome(MouseEvent mouseEvent) {...}

    private void loadTeam(){...}

    public void createNewCampaign(){...}

    private boolean checkFields(){...}

    //given i highlights the i° button
    public void highlightsButton(MouseEvent mouseEvent){...}

    private void reset(){...}

    @Override
    public void initialize(URL location, ResourceBundle resources) {...}

    public void setReference(HomeController homeController) {...}
}
```

- TeamViewController.java
  It is in charge of showing all the teams the user belongs to, the members of each and managing the functionality of adding a new team member.

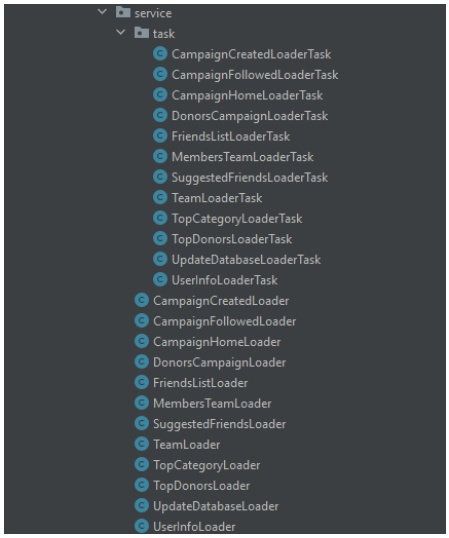
```
public class TeamViewController implements Initializable {

    @FXML
    Pane frostPane,containerCentralHome,containerOtherInterfaces,memberDetailsPane,listOfFriendsToAdAsMemberPane, newTeamPane;
    @FXML
    HBox boxTeams,boxMembers;
    @FXML
    ProgressIndicator progressIndicatorTeams, progressIndicatorMembers, progressIndicatorListOfFriendsToAdd,progressIndicatorNewTeam;
    @FXML
    Label closeDetailsMemberLabel,addMemberButton,labelTeams,labelTitleNewTeam;
    @FXML
    ImageView closeDetailsMemberImage;
    @FXML
    ListView listViewOtherTeams;
    @FXML
    ImageView addMemberImage;
    @FXML
    VBox boxListOfFriendsToAdd;
    @FXML
    Button createNewTeamButton;
    @FXML
    TextField areaNameOfNewTeam;
```

```java
    private ArrayList<TeamController> teamsController; //array with the controllers of all the teams
    private ArrayList<FriendToAddController> friendsList; //array of All the controller of friends of users
    private TeamController teamSelected; //this variable contains the position (in teamsController) of the selected team

    private UserInfo userInfo;
    private HomeController homeController;
    private static final Effect frostEffect = new BoxBlur( width: 100, height: 100, iterations: 3);

    public void setUserInfo(UserInfo userInfo){...}

    public void loadTeams(){...}

    public String getUsernameOfCurrentUser(){...}

    public void addTeam(TeamController teamController) { teamsController.add(teamController); }

    public void clearTeam() { teamsController.clear(); }

    public void setHomePanel(Pane containerCentralHome, Pane containerProfile){...}

    public void removeMemberFromBox(TeamController teamController, int pos){...}

    public void backToHome(MouseEvent mouseEvent) {...}

    public void showPopUp(String mex){...}

    public void enableIndicatorForListOfAddableFriends() {...}

    public void disenableIndicatorForListOfAddableFriends() {...}

    public Pane getFrostPane() { return frostPane; }

    public HBox getBoxMembers() { return boxMembers; }

    public ProgressIndicator getProgressIndicatorMembers() { return progressIndicatorMembers; }

    public Pane getMemberDetailsPane() { return memberDetailsPane; }

    public ListView getListViewOtherTeams() { return listViewOtherTeams; }
    public void setTeamSelected(TeamController teamController, boolean isRemoving){...}

    @Override
    public void initialize(URL location, ResourceBundle resources) {...}

    public void closeMemberDetailsPlane(MouseEvent mouseEvent) {...}

    public ArrayList<String> getUsernameOfMembersOfSelectedTeam(){...}

    public long getIdOfSelectedTeam(){...}

    public void updateLabelTeams(){...}

    public void deleteTeamUI(TeamController teamController){...}

    public void openFriendsList(MouseEvent mouseEvent) {...}

    public void addFriendToTeam(String username, String urlImmagineProfilo){...}

    public void clearBoxMembersUI(){...}

    public void addFriendToList(FriendToAddController friendToAddController){...}

    public void clearFriendsList(){...}

    public void hideAddFriendButton(){...}

    public void addNewFriendToListOfSelectedTeam(String username){...}

    public void openCreateTeamView(MouseEvent mouseEvent) {...}

    public void createNewTeam(MouseEvent mouseEvent) {...}

    private void showPopUpMessage(String s) {...}

    private void addNewTeamToUI(Team newTeam) {...}

    public void hideFrostEffect(MouseEvent mouseEvent) {...}

    public void setReference(HomeController homeController) {...}

    public void showPopUpOfExplorationMode() {...}

    public void destroy() {...}
}
```

- FriendsListController.java
  It is in charge of showing all the user's friends and managing their removal.

```java
public class FriendsListController implements Initializable {

    @FXML
    Pane containerCentralHome,containerOtherInterfaces;
    @FXML
    HBox boxFriendsList;
    @FXML
    ProgressIndicator progressIndicatorFriendsList;
    @FXML
    Label labelNumberOfFriends;

    private UserInfo userInfo;
    private ArrayList<FriendController> friendsList;
    private HomeController homeController;

    public void setReference(HomeController homeController) { this.homeController = homeController; }

    public void loadFriendsList(){...}

    public void setUserInfo(UserInfo userInfo){...}

    public void updateNumberOfFriendsLabel(int c){...}

    public String getUsernameOfUser(){...}

    public void addFriendToList(FriendController friend){...}

    public void addFriendToUI(Parent fxmlFriend){...}

    public void removeFriendFromUI(FriendController friendController){...}

    public void setHomePanel(Pane containerCentralHome, Pane containerProfile){...}

    public void backToHome(MouseEvent mouseEvent) {...}

    @Override
    public void initialize(URL location, ResourceBundle resources) {...}

    public void destroy() {...}

    public void showPopUpOfExplorationMode() {...}

    public void showPopUp(String mex){...}

    public void showPopUpMessage(String s) {...}
}
```

The use of a particular package, called ***service***, should be emphasized:



Thanks to this package it is possible to make the user experience of users as fluid as possible. In fact, most of the data uploads are done in the **background** so as not to block the UI.

Paragraph 15

Graphic Interface

Upon startup the user will be presented with the following screen:

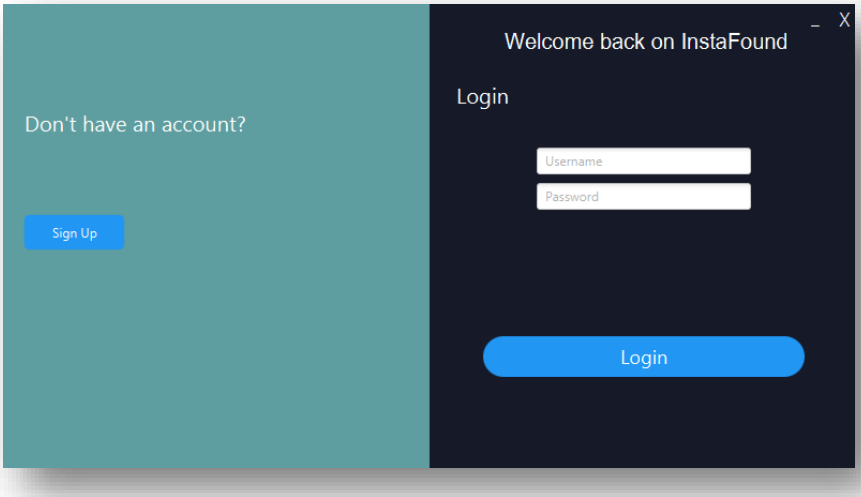


Figure 18: startup screen.

The user can log in if he already has an account or make a new registration in which he is encouraged to enter the username, the url of his profile image, the region, province (initials) and cap code of origin:

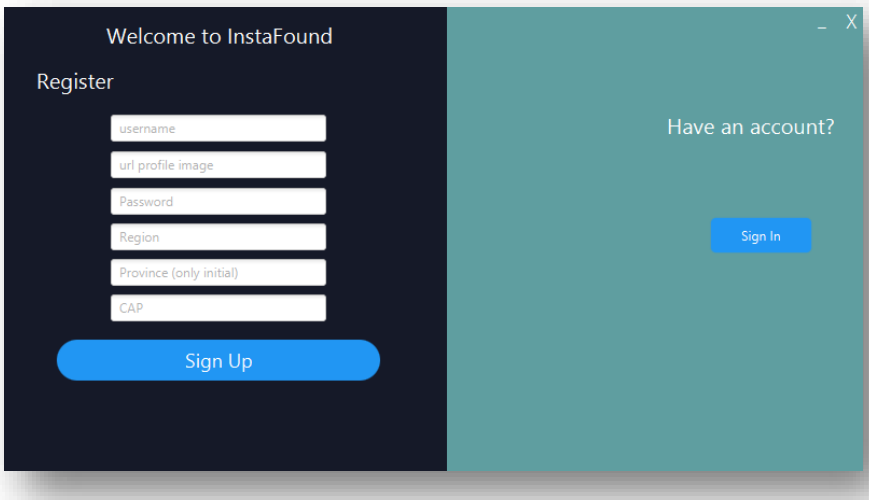


Figure 19: registration screen.

After login a new screen will appear to the user:

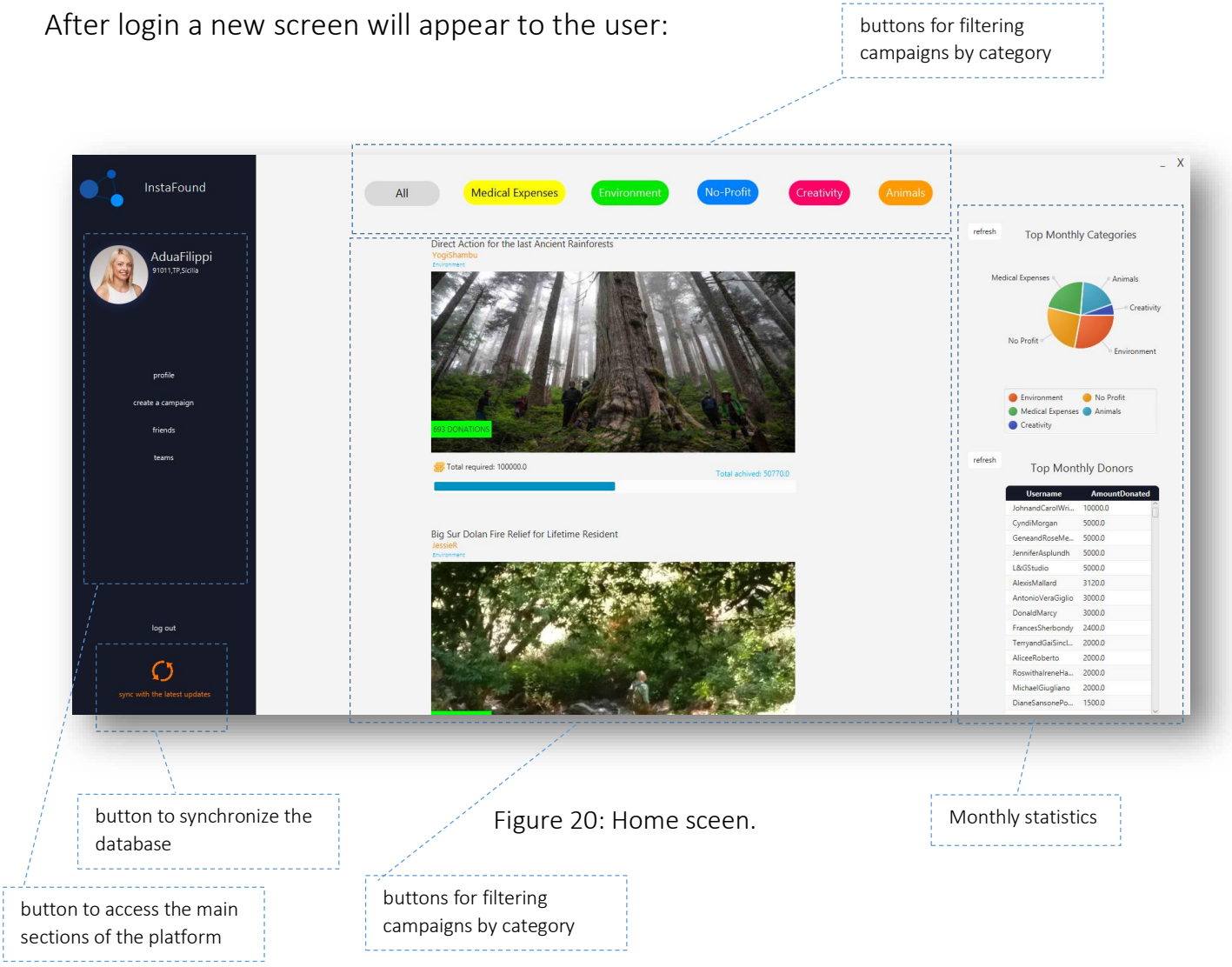


Figure 20: Home sceen.

On the home page, the user can scroll through the latest campaigns published by filtering (or not) by the 5 Instafound categories: Medical Expenses, Environment, No-Profit, Creativity, Animals.

For each campaign, only the basic information is shown, which summarizes it. Only 10 campaigns are loaded on the home page. If the user scrolls to the end of the 10 campaigns, a "load more" button will (if clicked) load another 10 campaigns of that category. The column on the right presents the two monthly statistics: the best categories and the list of the best donors. On the left there is the main panel to move to the other 4 main sections of InstaFound. At the bottom there are two special buttons, the first to log out, which redirects to the screen in figure X, and finally the button to synchronize the database:

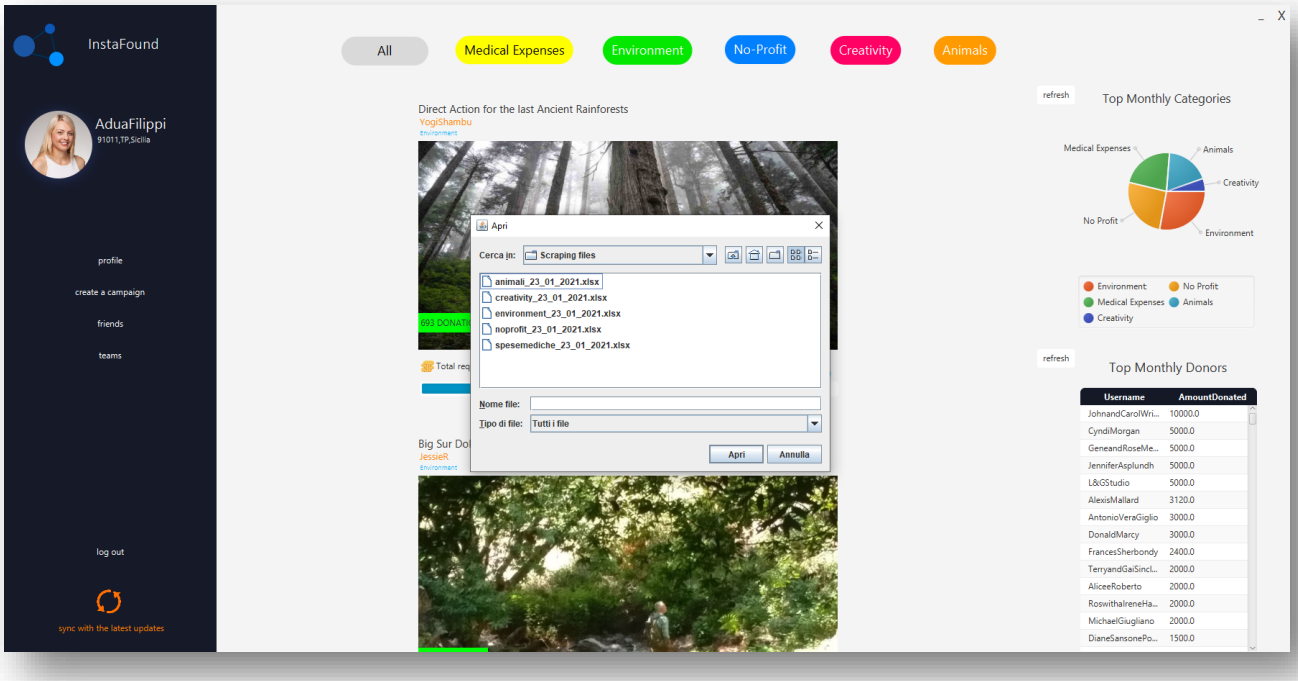


Figure 21: screen to choose the file to upload.

Once chosen, the loading of the new data will start:

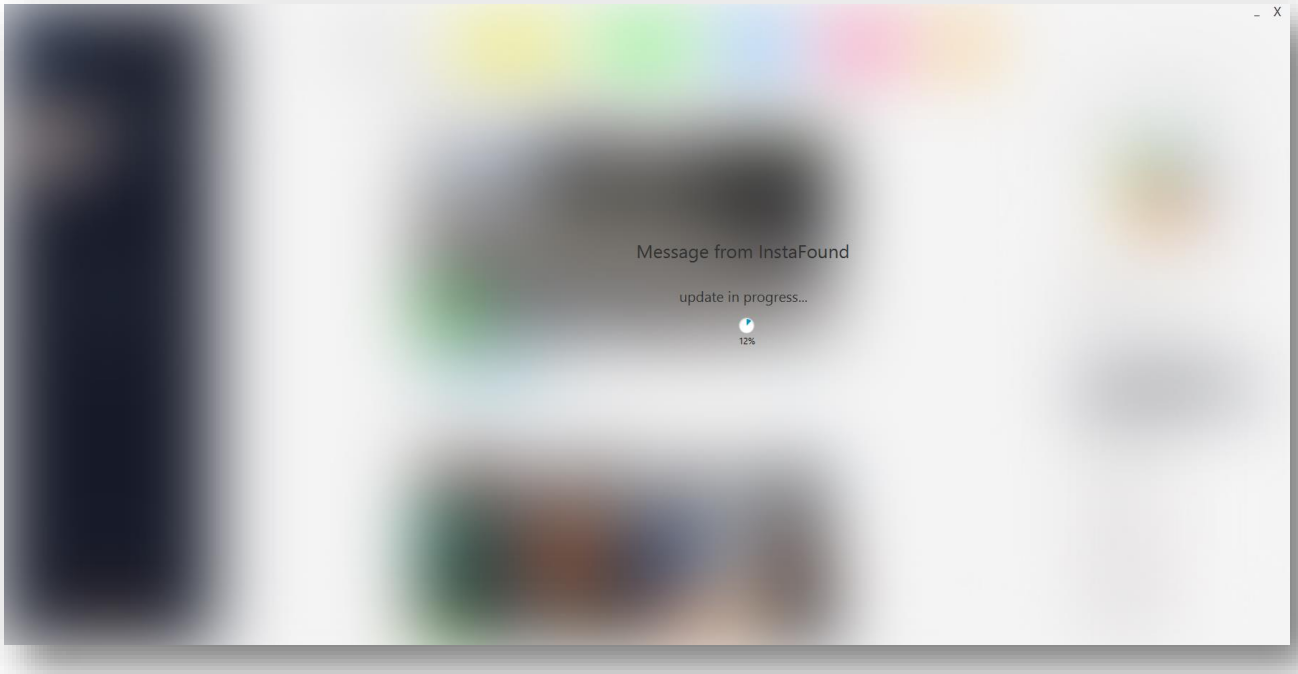


Figure 22: updating screen.

When the user clicks on one of the campaigns shown on the home page, a detail screen will open:

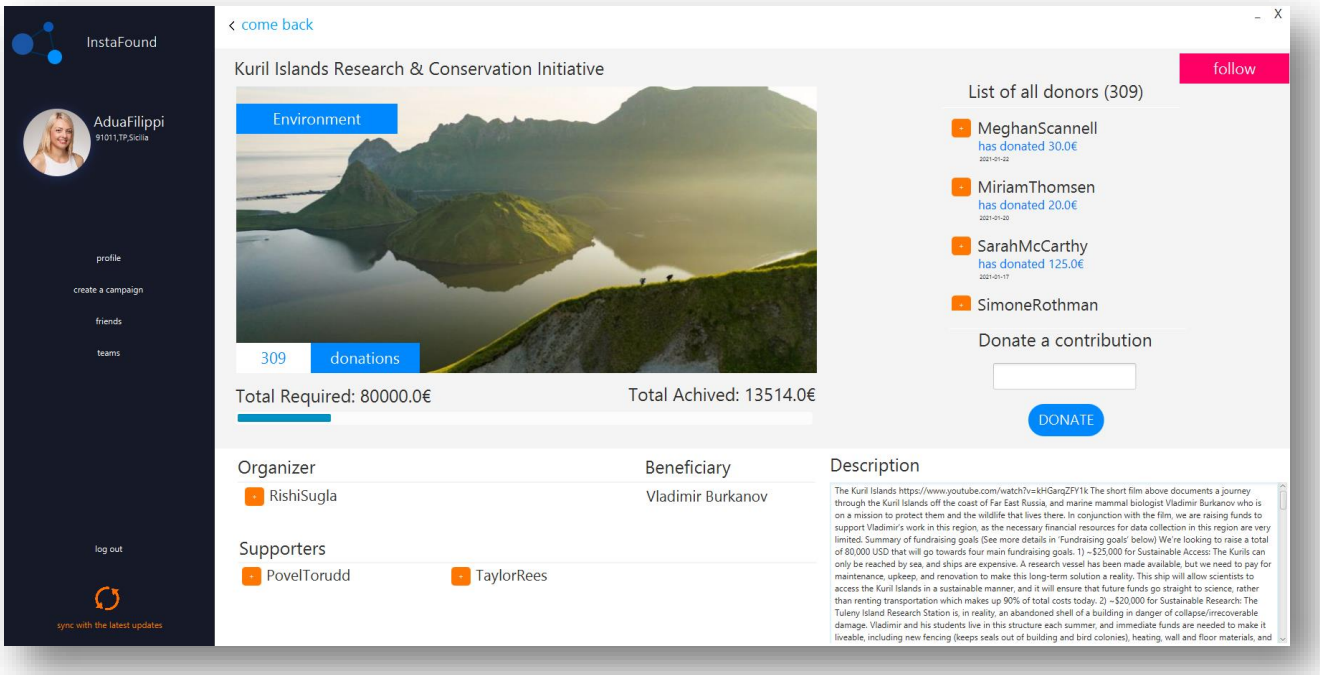


Figure 23: campaign details screen.

The campaign detail screen shows its title, image, the category to which it belongs, the number of donations received, the amount requested and the amount received and the relative progress shown by the progressive bar and the description. Also shown are the organizer and the members of the team supporting the campaign.

On the right there is a list of all donors with their donation timestamp and amount donated.

Each user shown (organizers, members and donors) have a square on the left side which, if covered by the mouse, shows the possibility of adding that user to their circle of friendship. There is no "friend request" here.

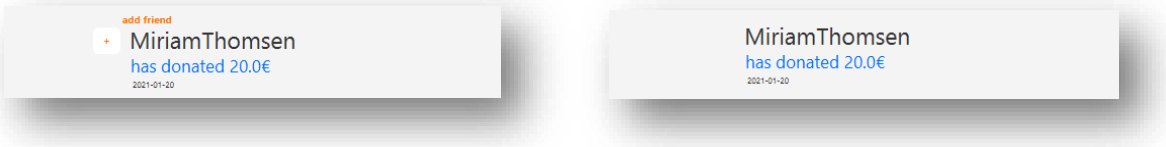


Figure 24: adding a donor to your friends list (before and after).

At the bottom, a "donate" button allows you to donate:

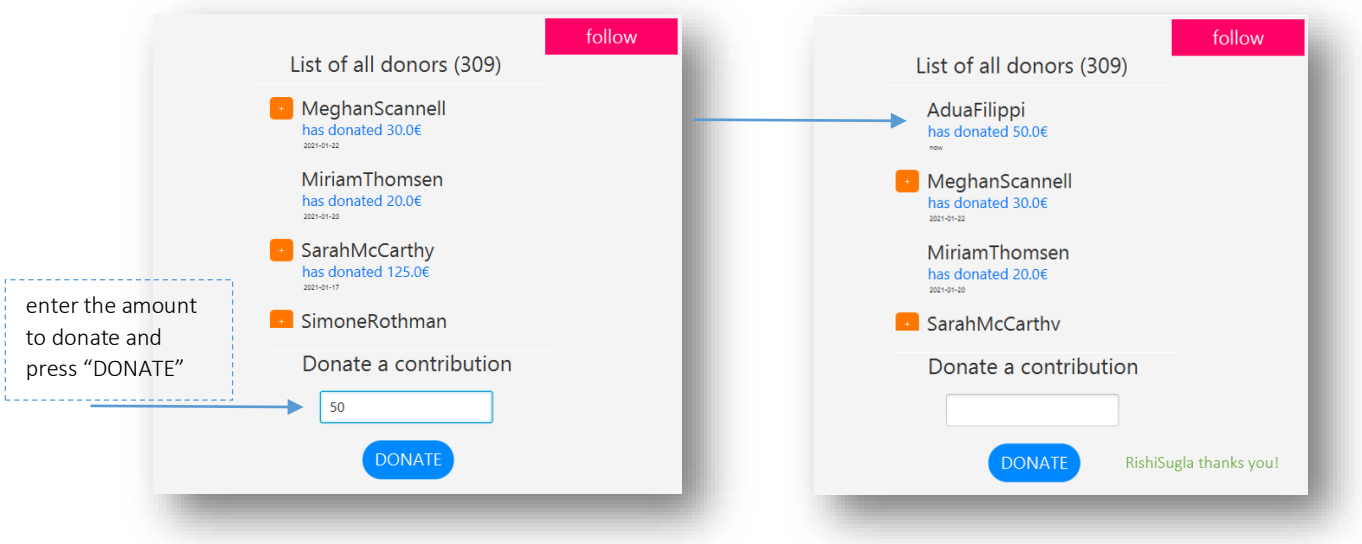


Figure 25: example of a donation of 50€.

The first section of the panel on the left is that of the **profile**:

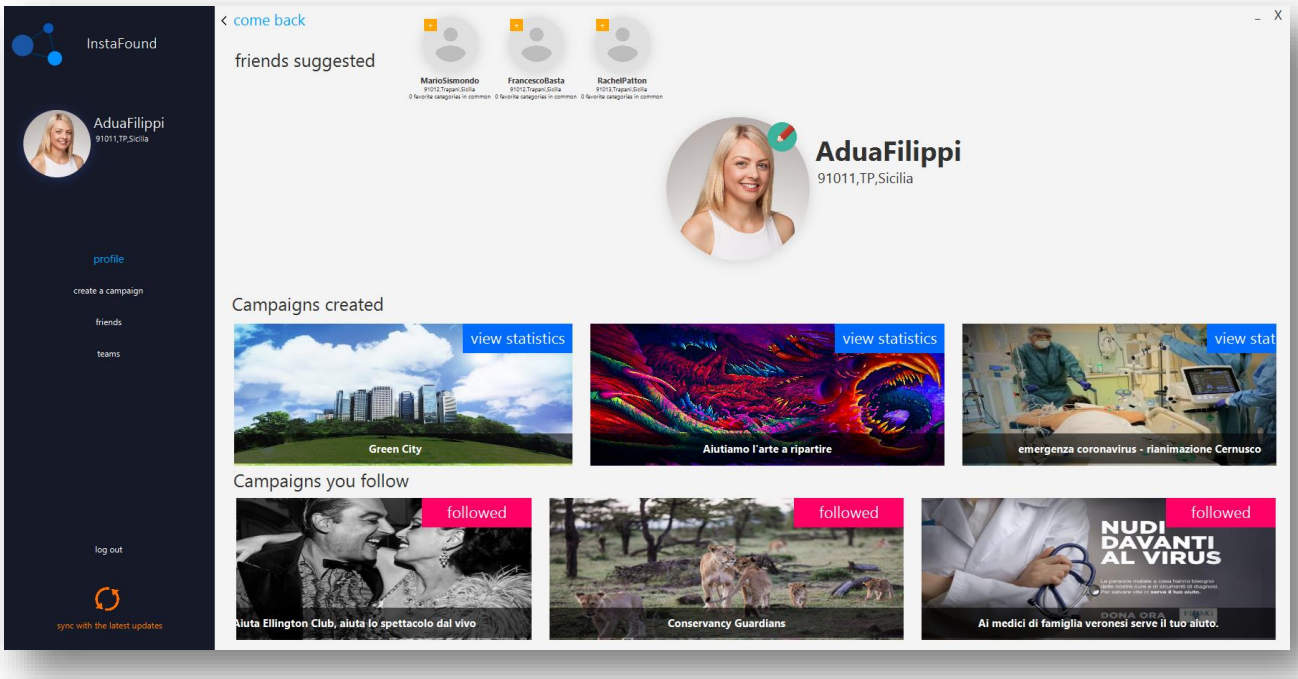


Figure 26: profile screen.

The information of the current user is shown in the center. By clicking on the icon in the corner of your profile picture it's possible to change profile picture:

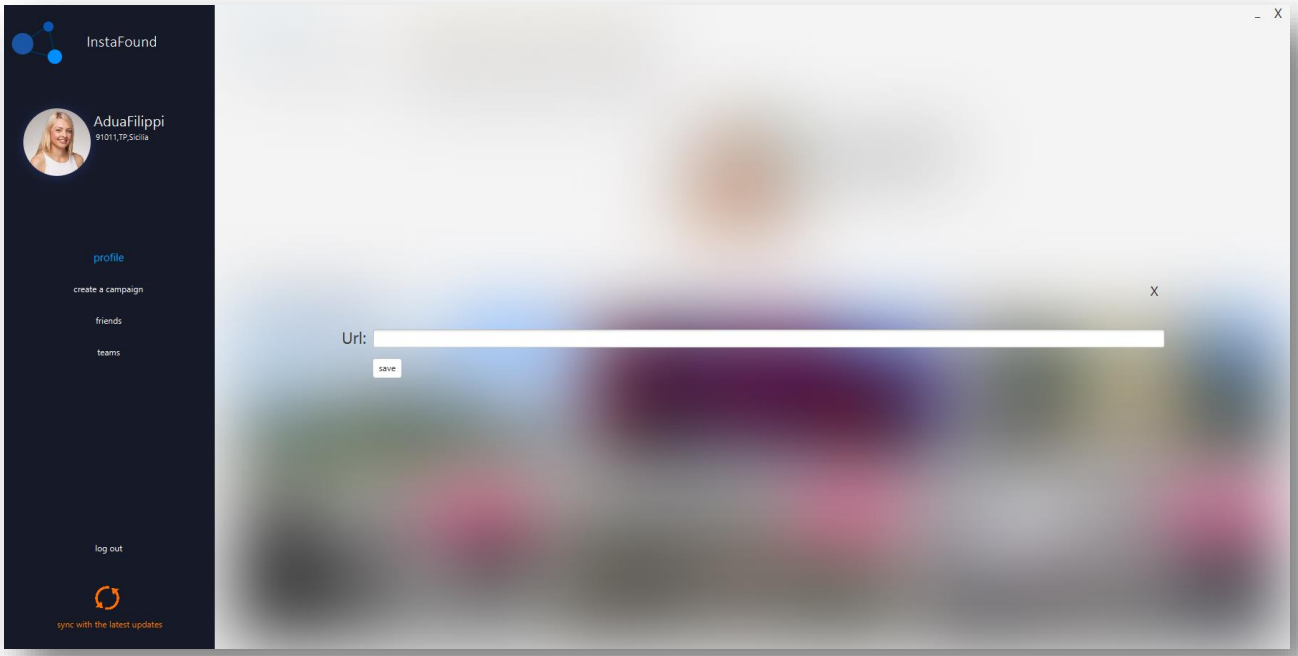


Figure 27: area to insert the new url of profile image.

Above, instead, new users are suggested to be included in their circle of friendship. The suggestion introduces users closest to and with the greatest interests in common in terms of campaign types with the current user. As with the donors in figure x, the orange square allows, if clicked, to add to friends. The algorithm provide almost always different users from time to time.

The campaigns created and followed are shown below. It is possible to unfollow a campaign by clicking on the relative button which will show the word "unfollow" when the mouse passes over it:

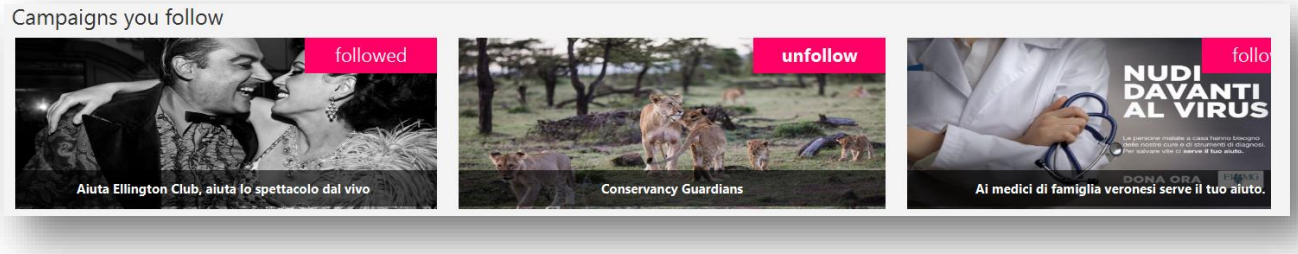


Figure 28: example of unfollowing a campaign.

For each campaign created or followed it is possible, if it's clicked in the relative image, to open the relative details as seen in figure x.

regarding the campaigns created, the user has an additional feature which is to see the statistics:

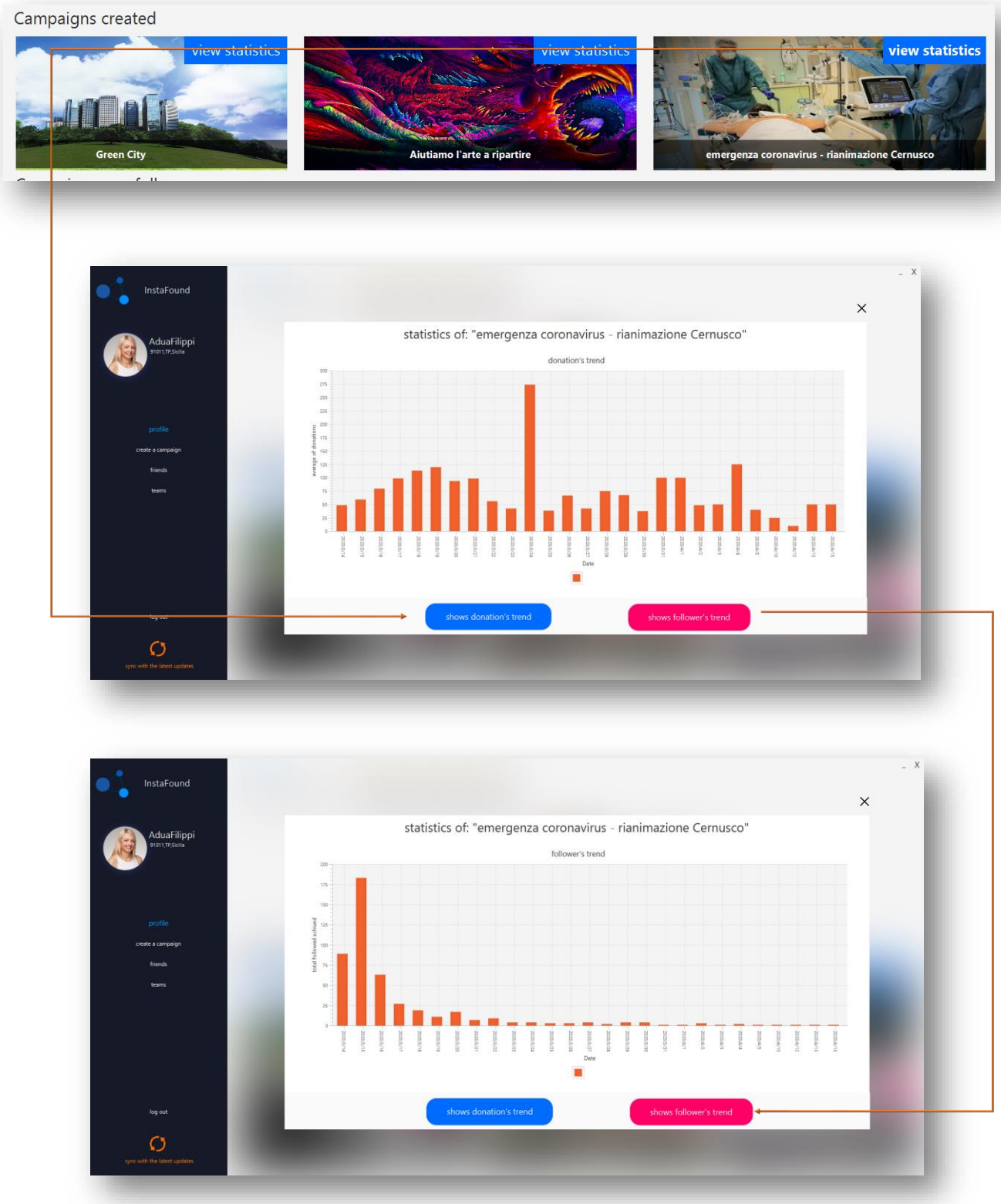


Figure 29: trend of donation and followers for a specific campaign.

The second main section is the one concerning the **creation of a new campaign**:

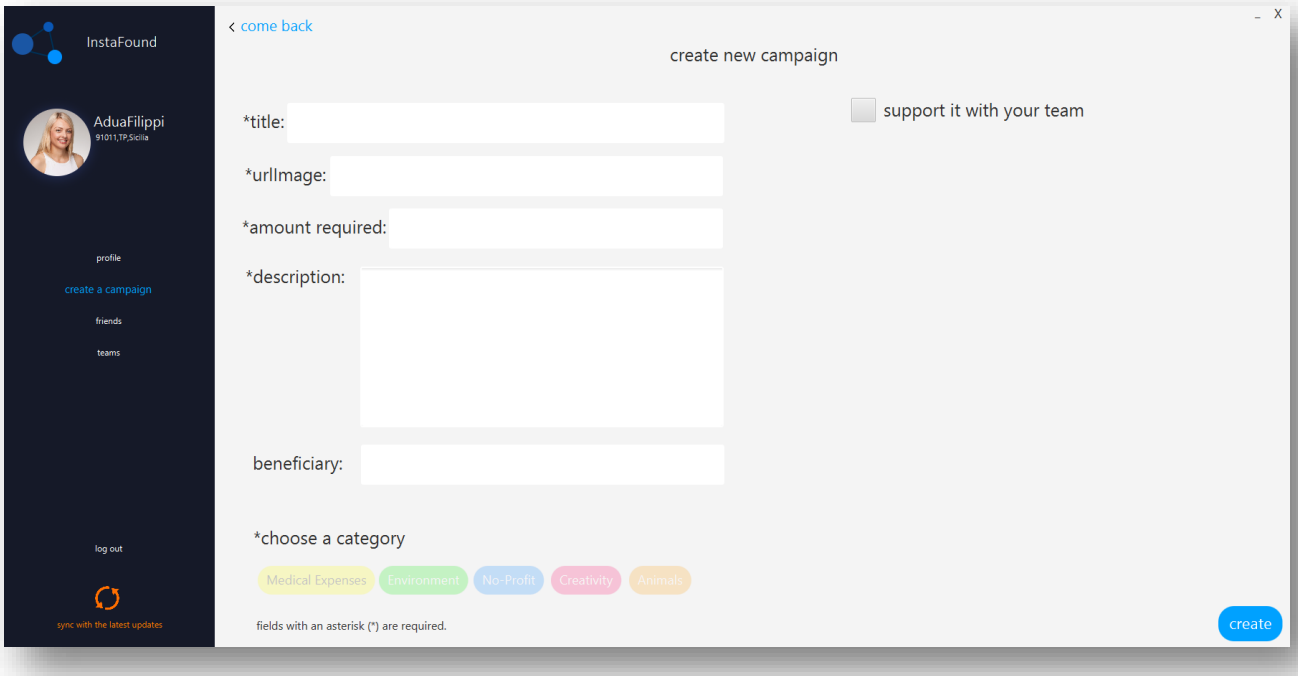
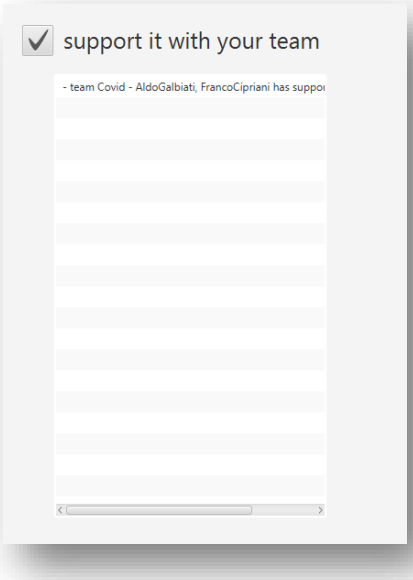


Figure 30: creation of a new campaign screen.

All fields are required except for the beneficiary and the team choice as it is possible to support the campaign without a team. Alternatively, selecting a team expects to show all the membership teams:



A list of the teams he created is shown. All current members and supported campaigns are also shown for each team.

If there are errors (empty, unselected fields or network errors) the platform warns the user with a message.

Figure 31: list of membership teams.

The third most important section is the one concerning the **list of user friends**:

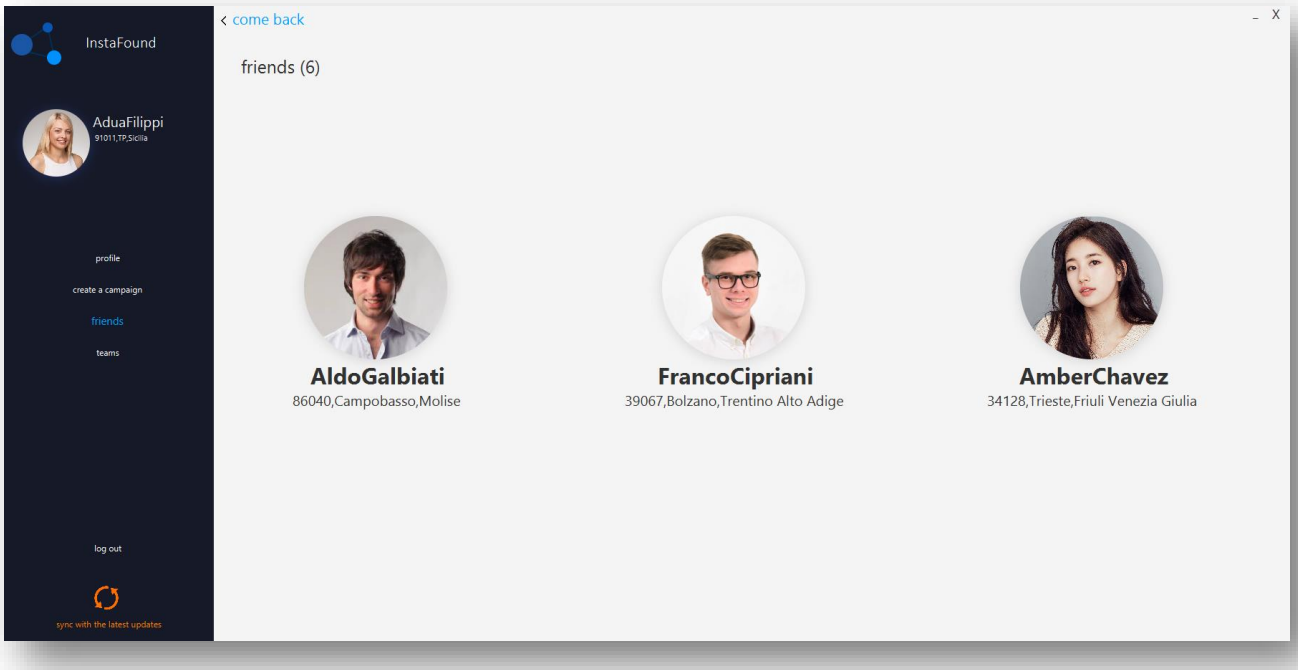


Figure 32: list of friends screen.

In addition to seeing information about each user, it's also possible delete a user from your circle of friends passing over his image:

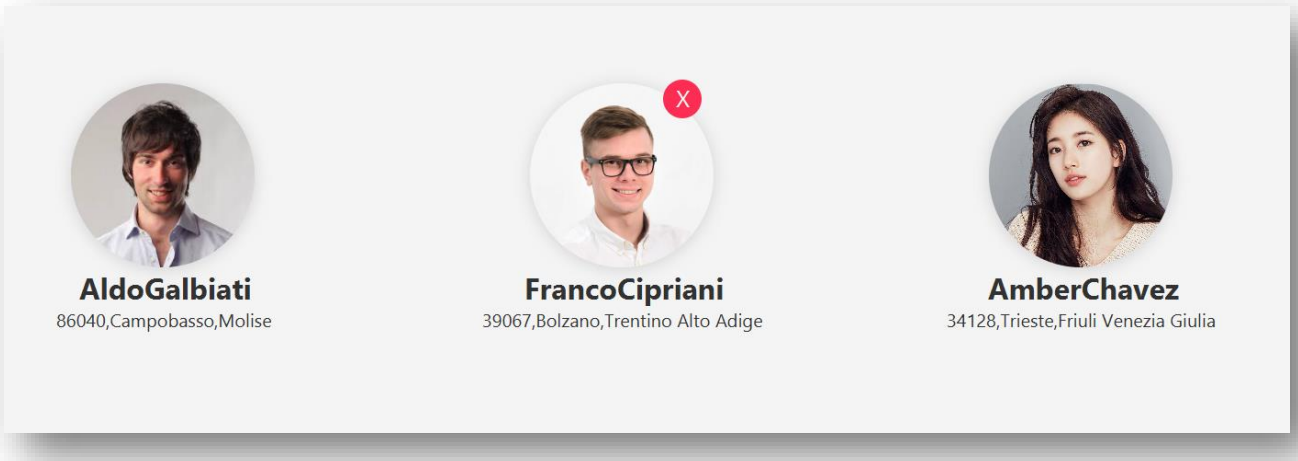


Figure 33: removing a user from friends list.

The fourth and last most important section is that of **teams**:

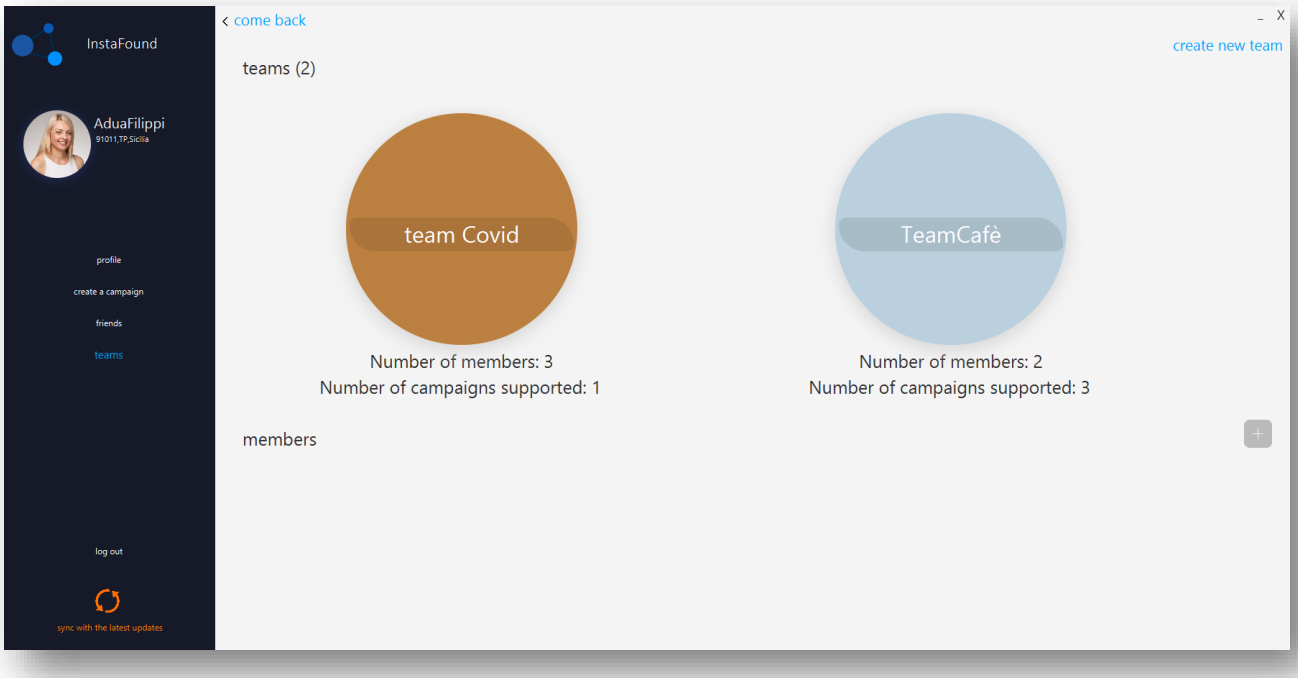


Figure 34: teams screen.

The screen shows both the teams the user has created and those have been added to. It is possible to remove a team only if the user is the organizer, otherwise it's possible only to leave it:

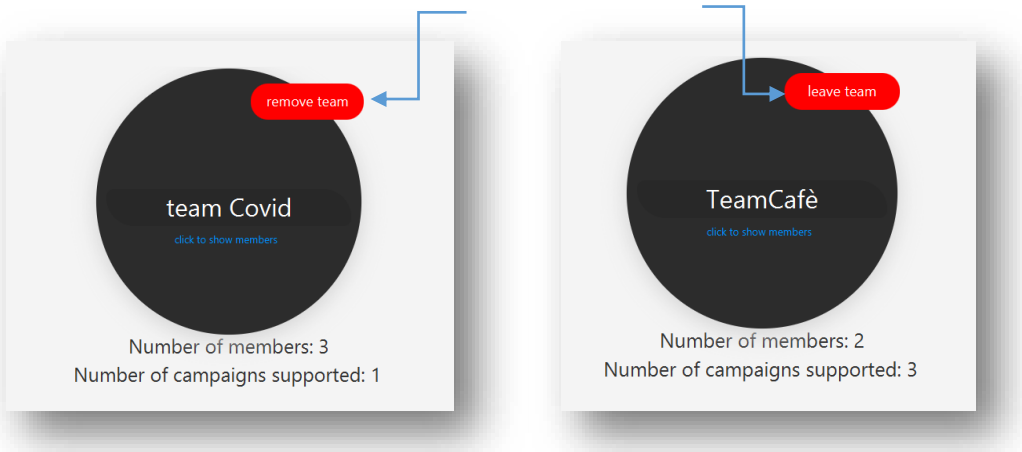


Figure 35: remove (if organizer) or leave it (if simple member).

It is possible to create a new team by selecting the button on the right-top and choose the name:

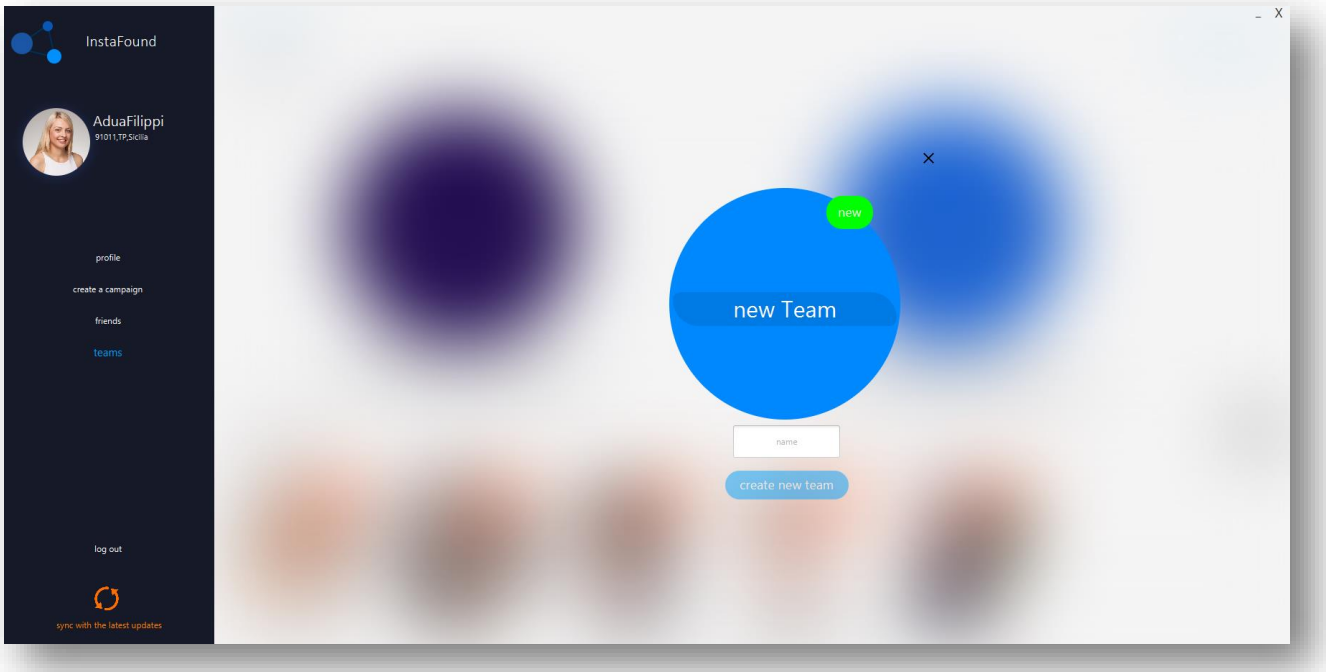


Figure 36: creation of a new team.

By clicking on a specific team it's possible to see all the members. Each member has a label that indicates whether they are organizer or a simple member:

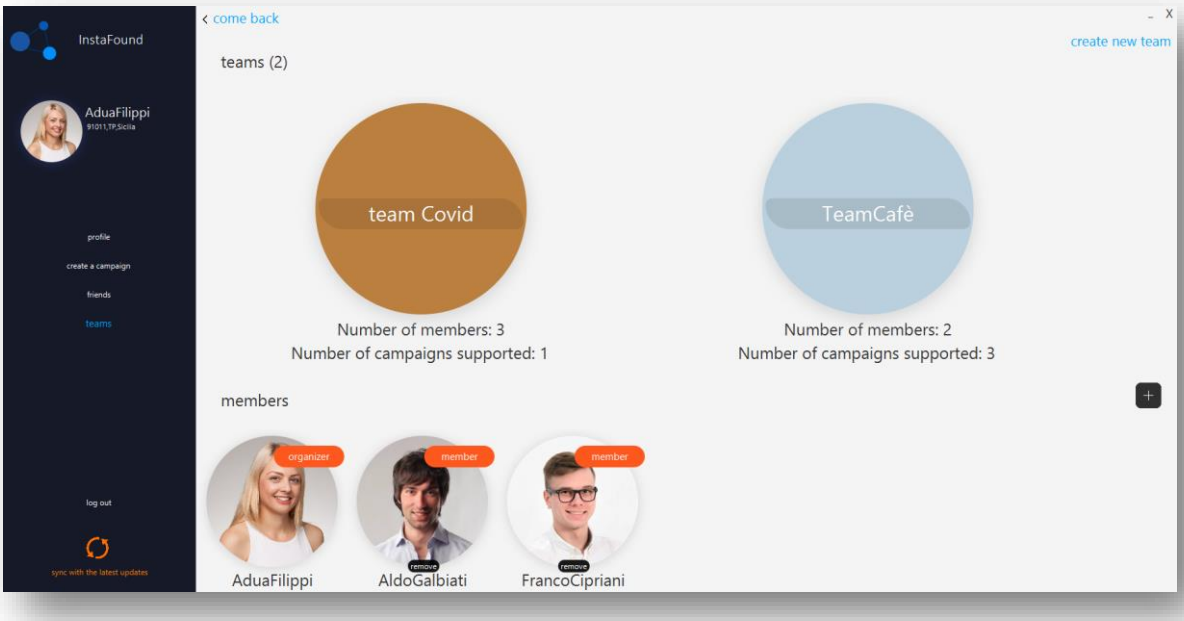


Figure 37: example of a team of which the user is the organizer.

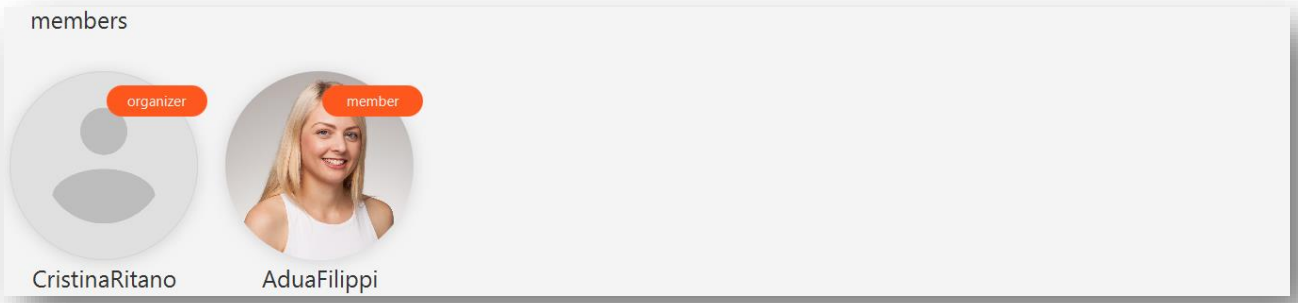


Figure 38: example of a team of which the user is a simple member.

For each member it is possible to see the teams and other campaigns in which they participate by clicking on their image:

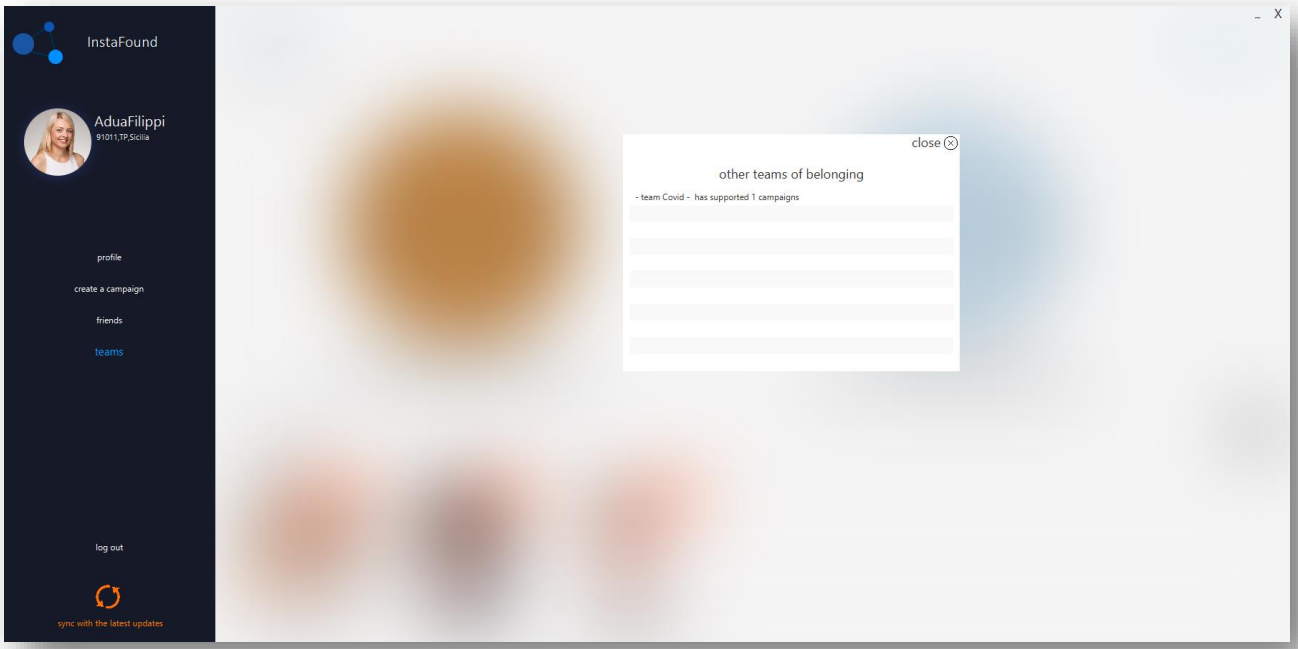
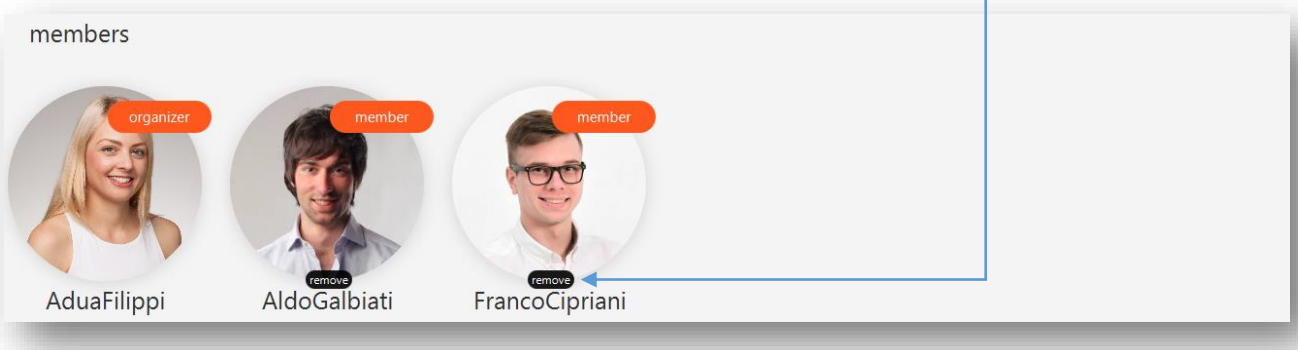


Figure 39: the member only belongs to a teamCovid. Same as shown before.

If the current user is the organizer of the team he has special privileges, such as deleting a member:



Another privileged function is to add new members:

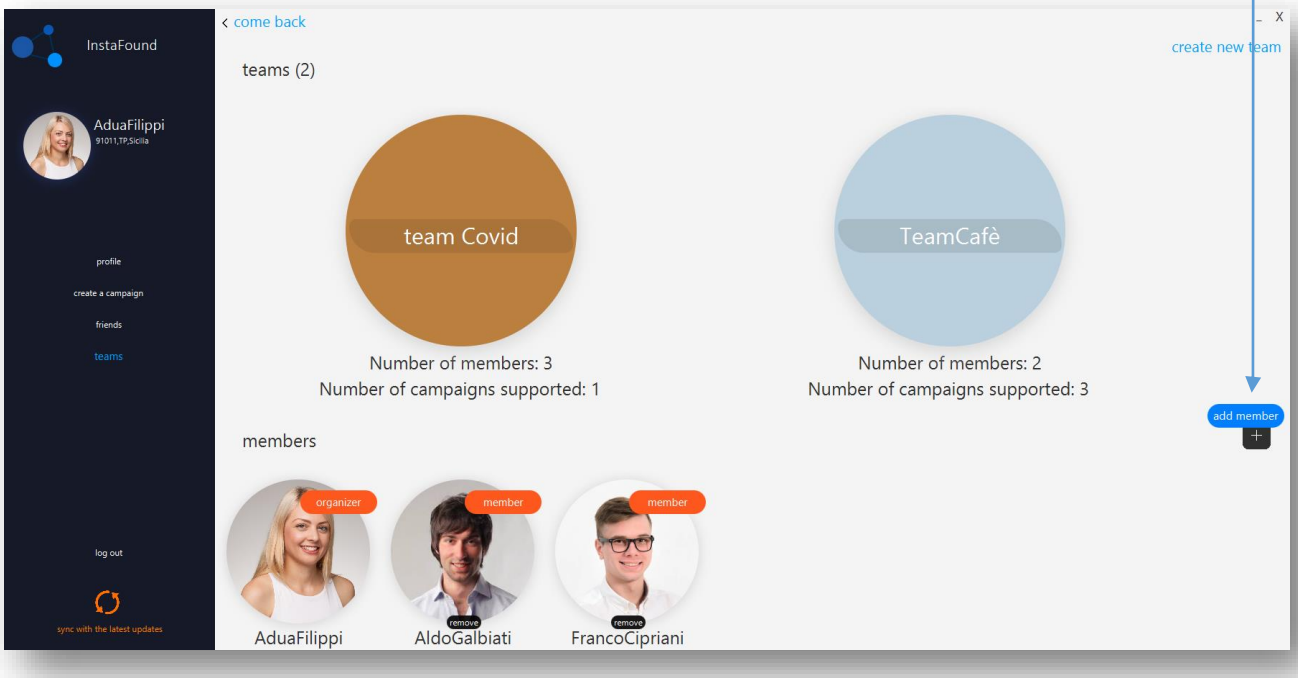


Figure 40: “add member” button to add new member to the team.

A list of the current user's friends will only show those that can be added, i.e. those not currently belonging on the team:

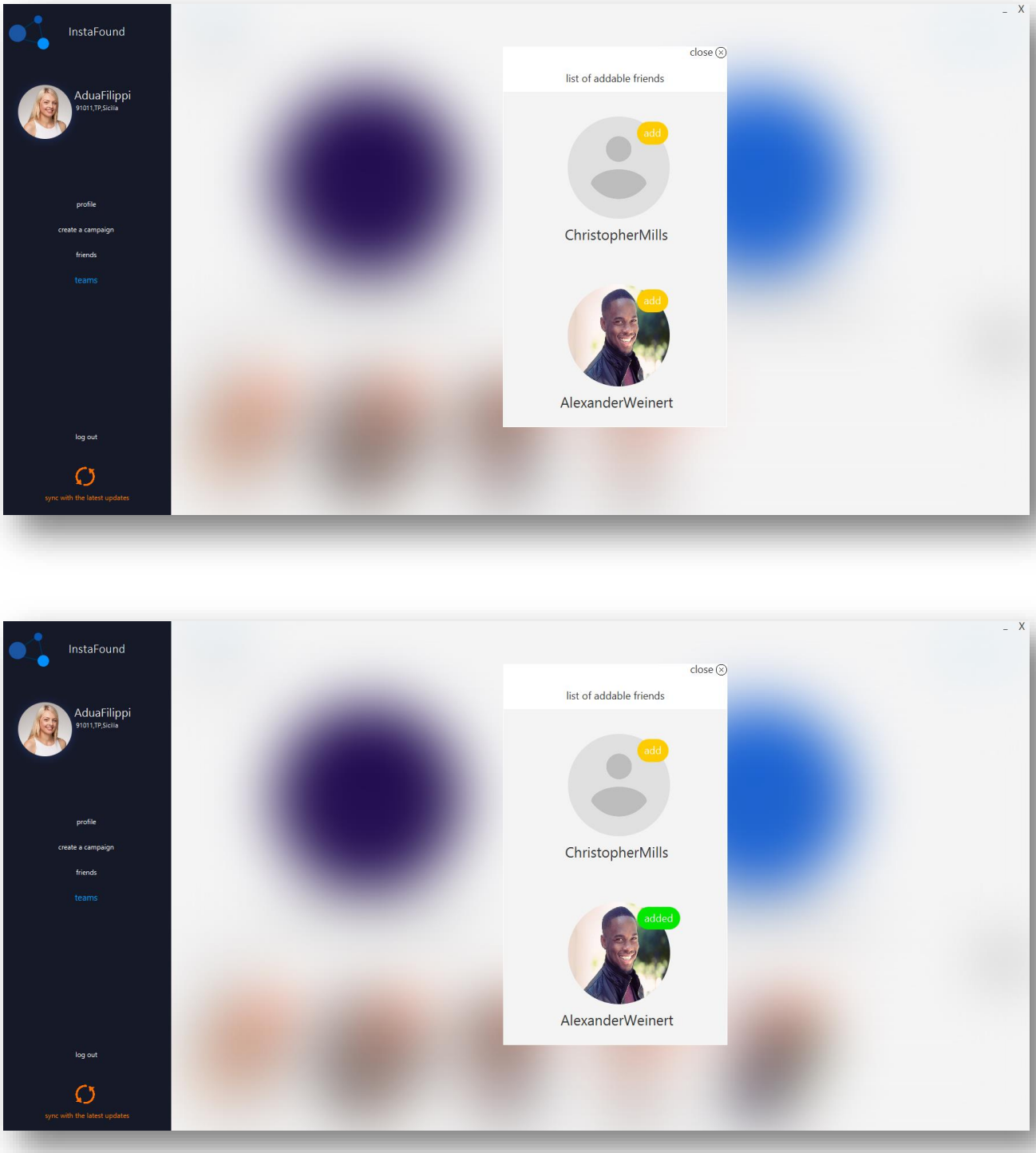


Figure 41: before and after the addition of new member.