UNIVERSITY OF PISA

School of Engineering

Master of Science in Computer Engineering

Large Scale And Multi-Structured Databases

PROJECT DOCUMENTATION

PSEUDOSTACKOVERFLOW: A JAVA + MONGODB + NEO4J BASED APPLICATION IMPLEMENTING A SIMPLE PROGRAMMING FORUM

WORKGROUP:

Leandro Giacomazzo

Iacopo Pacini

Gaetano Niccolò Terranova

ACADEMIC YEAR 2021/2022

Contents

1. Introduction

PseudoStackOver is a programming forum in which professional programmers or simply enthusiasts can publish programming/IT related questions. All questions are public, everyone can browse them but only registered users can answer and mark with positive or negative vote already existing answers. Every user is equipped with a personal profile containing a brief description uploaded by the user, her location and a profile picture. Users can "follow" each other and periodically check if one of their follower posts new questions or simply answers to a post. The complete code can be found at this address: `https://github.com/gaetanoterra/progettoLSMD`

1.1 Functional Requirements

This section shows the main functional requirements of the application:

- The platform allows to be navigated in three types of modes: unregistered mode, register mode, admin mode.
- Any user should be able to browse the stored posts
- A registered user should be able to remove a post of his own
- A registered user should be able to remove an answer of his own
- A registered user should be able to answer any stored post
- A registered user should be able to vote an already existing answer
- A registered user should be able to browse his followed users list
- A registered user should be able to Enter and update their personal information

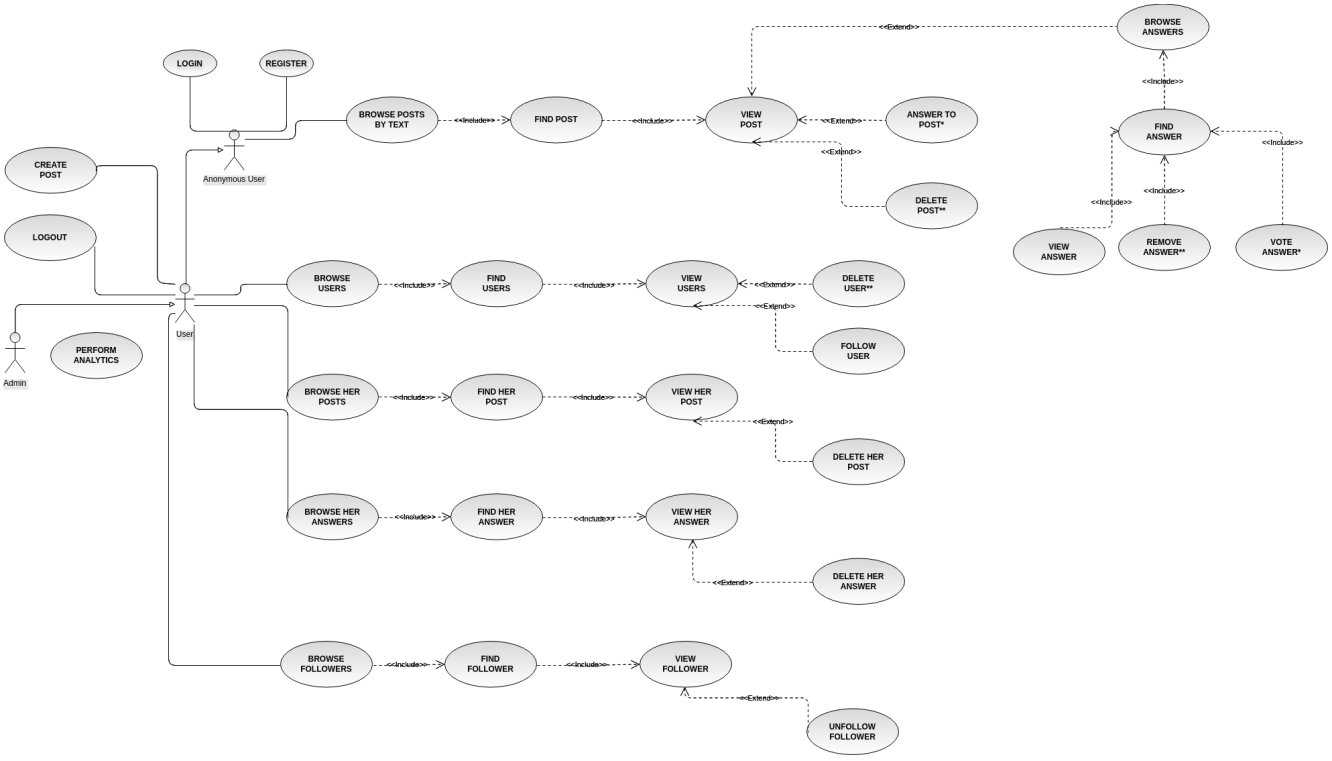
1.2 Non-Functional Requirements

This section shows the main non-functional requirements of the application:

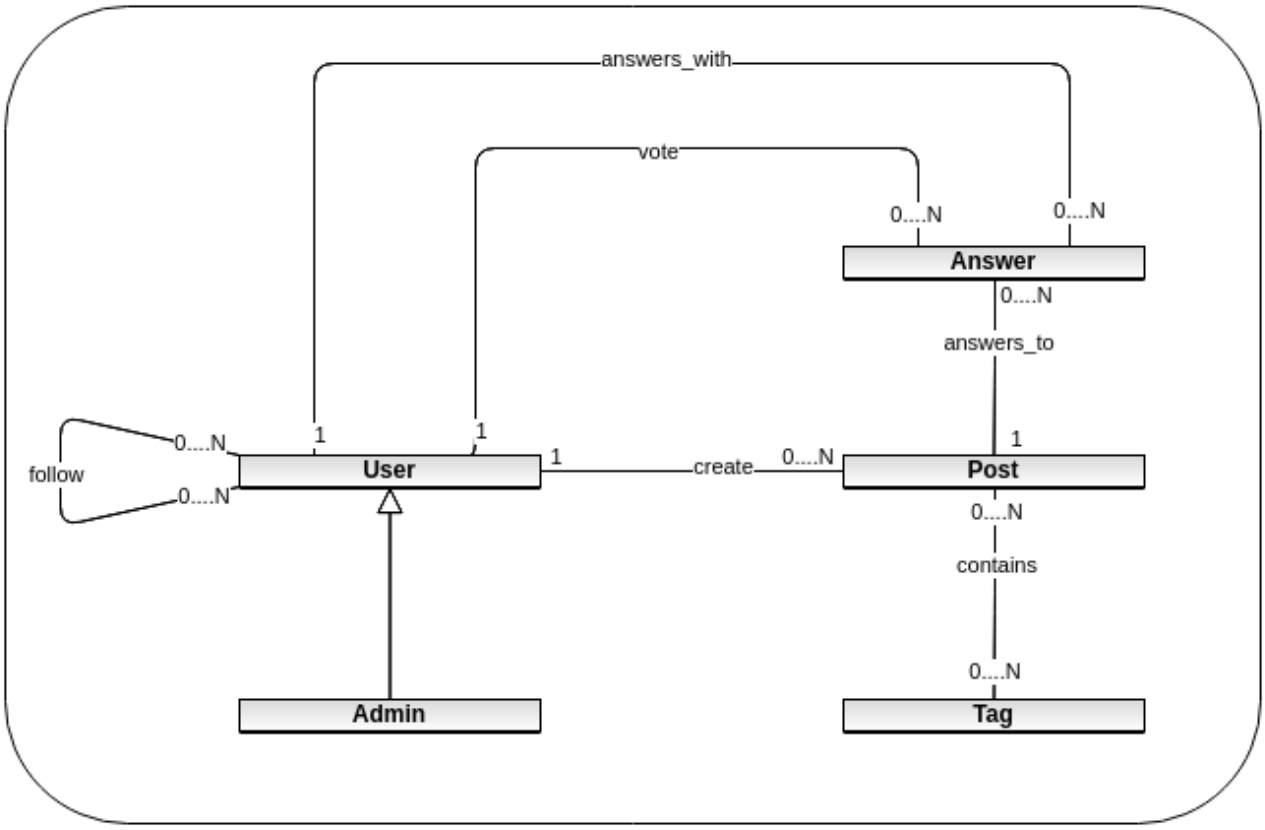
- The application must have a low response time.
- The application must guarantee data availability.
- The application must ensure prevention against server crashes thanks to replicas.
- Admins of the application periodically monitor the behaviour of Users in order to guarantee that they comply to the Terms & Agreements.

2. Design

2.1 UML Use Cases Diagram



2.2 UML Analysis Class Diagram



3. Implementation

3.1 Document Database

The document part of the application is implemented using MongoDB, a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. The document database contains two collections: Users and Posts listed below.

Posts document structure

```
{
    _id: ObjectId,
    AnswerId: string,
    CreationDate: long,
    ViewCount: int,
    Body: string,
    Title: string,
    Tags: [ string ],
    DisplayName: string,
    Answers:
        [{
            Id: string,
            CreationDate: long,
            Body: String,
            OwnerDisplayName: string,
            Score: int
        }]
}
```

The latter document contains all the information related to a question posted on the forum, in particular:

- **GlobalId** computed as

  SHA256({ Title | CreationDate })

  It allows to identify univoquely a post among the two databases.
- **CreationDate** is a long value representing the date and the time in which that particular question is uploaded into the forum;
- **ViewCount** field keeps track of the overall times in which that post is read;
- **Body** contains the actual question, namely its text body;
- **Title** contains the title of the question;
- **Tags** is a list of tag names that the a user encloses to the post itself;
- **DisplayName** is the username of the author;
- **Answers** is a list of documents containing creation date, text body, author username and the algebraic sum of the received votes. The choice of embedding this collection is due to the fact that the answers to a post are generally limited to a few units. Moreover, this setting allows to populate entirely the FullPostInterface(one of the most requested) with a single find() query on _id field.

Users document structure

```
{
    _id: ObjectId,
    Reputation: int,
    CreationDate: long,
    DisplayName: string,
    WebsiteUrl: string,
    Location: string,
    AboutMe: string,
    ProfileImageUrl: string,
    Password: string,
    followerNumber: int,
    followedNumber: int,
    IsAdmin: boolean
}
```

- **Reputation** The algebraic sum of the Votes given to the answers given by the user itself;
- **CreationDate** is a long value representing the date and the time in which that particular user joined the forum;
- **DisplayName** is the username of the user;
- **WebSiteUrl** user personal web-site URL;
- **Location** where the user is located;
- **AboutMe** contains the title of the question;
- **ProfileImageUrl** is a list of tag names that the a user encloses to the post itself;
- **Password** User profile password;
- **followerNumber** Amount of user following that particular profile;
- **followedNumber** Amount of user followed by that particular user;
- **isAdmin** optional field indicating that the user has administrator priviledges;

3.1.1 Queries Analysis

Read Operations		
Operation	Expected Frequency	Cost
Retrieve complete informations about a post by Post title/body	Very High	Medium (text search)
Retrieve complete informations about a post by Post title/body	Very High	Low (search by _id)
Retrieve complete informations about a User given his(her) username	High	Medium (text search)
Given a tag, find the users who answers gained the highest score with their answers.	Low	High (aggregation)

Write Operations		
Operation	Expected Frequency	Cost
Insert/remove a new Answer/Post/User	Low	Average (document add/remove)
Update Answer Score	Average	Low (1 attribute write)
Update user data	Very Low	Low (few attributes write)

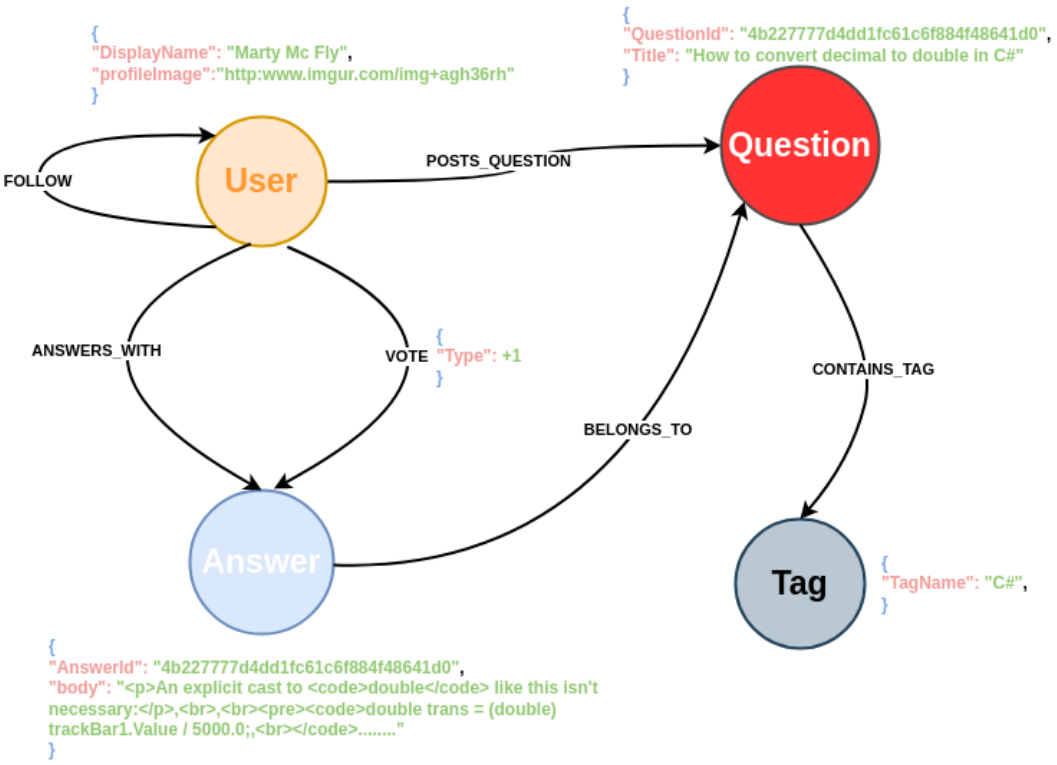
All the assumption made previously are based on the fact that a question(or an answer) is posted only once and accessed many times by both registered and anonymous users, the same holds for Users data. It can be concluded that the queries are predominantly read intensive, for this reason indexes to speed up read operations (at the expense of write operations) are added.

3.1.2 Queries Implementation

Query	Description
db.Posts.aggregate([{$match: {"Tags":"??"}}, {$unwind: "$Answers"}, {$match: {"Answers.DisplayName":{$ne:null}}}, {$group: { _id: "$Answers.DisplayName", totaleRisposteUtente:{$sum:"$Answers.Score"}} }, {$sort: {"totaleRisposteUtente": -1 } }])	This query allows to retrieve the most expert users on a given tag. Namely the ones whose answers gained he highest score
db.Posts.find({ $text:{$search: "??"}, $or:[{Title: {$regex: /.*??.* /i}}, {Body:{$regex: /.*??.* /i}}] }, { _id:1, Title:1, Tags:1, numberOfAnswers:{$size:"$Answers" }, ViewCount:1, },);	This query is a crucial feature for the forum, it is one of the most used, it can be exploited by both registered andanonymus users.
db.Users.find({DisplayName: "??"})	It allows to retrieve all the question asked by a user, it exploits an index on the username.

The most critical operations, as previously stated, are the log-in operation and the search by text. For this reason it has been chosen to create a compound text index on the fields Title and Body in the Posts collection and a text index in the displayName field in the Users collection. The compound index in all the benchmarks boosted the query performance at least of a factor ten. The index on the Users collection instead caused an even greater speed-up of two orders of magnitude in all benchmarks.

3.2 Graph Database



3.2.1 Nodes

- **User**, representing the users registered within the application, having as attribute its own username
- **Question**, representing a question asked by a user with its global id and its title. The question body is kept in the document db.
- **Answer**, representing an answer given to a particular question
- **Tag**, An unambiguous container for a particular tag.

3.2.2 Relationships

- **User → ANSWERS_WITH → Answer** which connects an answer with its own author. This relationship has no attributes.
- **Answer→ BELONGS_TO → Question**, which connects a question with its own author. This relationship has no attributes.
- **Question→ CONTAINS_TAG → Tag**, which connects a question with the corresponding tag Entity. This relationship has no attributes.
- **User → FOLLOW → User**, which represents a user following another in the application, and is created/removed when a User adds/removes another User from his followed users list. This relationship has no attributes.
- **User → POSTS_QUESTION → Question** which represents a user uploading a question on the forum, and is created/removed when a User posts/removes a question. This relationship has no attributes.
- **User → VOTE → Answer**, which represents a user adding a positive or negative vote to an answer. It contains an integer +1 or -1 representing a positive/negative vote as attribute.

3.2.3 Queries Analysis

Read Operations		
Operation	Expected Frequency	Cost
Retrieve answers posted by a given user along with the total score	Low	Medium (multiple reads and adjacencies)
Retrieve users who post questions containing the same tag of the questions asked given by a given user	Average	Medium (multiple reads and adjacencies)
retrieve the follower list of a given user	High	Low
retrieve the list of users followed by a certain user	High	Low
Retrieve the topics(in terms of tags) that sparked interests among the most followed users.	Low	High
retrieve the questions asked by the most followed users who have the highest number of answers	Low	High

Write Operations		
Operation	Expected Frequency	Cost
Insert/remove a new Question/Post/Answer	Low	Low
Insert a Vote relation	Average	Low

3.2.4 Queries Implementation

Query	Description
MATCH (topUsers:User)<-[f:FOLLOW]-(otherUsers:User) WITH topUsersdisplayName as t_us, count() as follower_no ORDER BY follower_no DESC LIMIT 10 MATCH (t:Tag)<-[c_tag:CONTAINS_TAG]-(q:Question) <-[b_to:BELONGS_TO]-(a:Answer) <-[an_with:ANSWERS_WITH]-(u:UserdisplayName:t_us) WITH u.displayName as top_users, follower_no, t.tagNames as tag_names, count() as tags_top_users ORDER BY tags_top_users DESC RETURN top_users, follower_no, tag_names, tags_top_users LIMIT 10	This query retrieves the questions asked by the most followed users who have the highest number of answers

MATCH (topUsers:User)<-[f:FOLLOW]-(otherUsers:User)
WITH topUsers.displayName as top_users, count() as folllower_no
ORDER BY folllower_no DESC LIMIT 10
CALL {
WITH top_users
MATCH (a:Answer)-[b_to:BELONGS_TO]->(quest:Question)
<-[pq:POSTS_QUESTION]-(u:UserdisplayName:top_users)
WITH u.displayName as t_users,
quest.Title as title,
count() as answers_no
ORDER BY u.displayName, answers_no DESC
LIMIT 3
RETURN title, answers_no
}
RETURN top_users, title, answers_no
	Retrieve the topics(in terms of tags) that
	sparked interests among the most followed users.

db.index.fulltext.queryNodes("displayname_fulltext_index", "???") YIELD node MATCH (node)-[:ANSWERS_WITH]->(a:Answer) <-[v:VOTE]-(uv:User) RETURN a.answerId as answerId, a.body as body, sum(v.VoteTypeId) as score ORDER BY score DESC	Retrieve answers posted by a given user along with the total score

MATCH (u2: User)-[:POSTS_QUESTION]->(:Question) -[:CONTAINS_TAG]->(t:Tag tagNames: "???") WHERE u2.displayName <> "???" RETURN distinct u2.displayName AS Username, u2.profileImage AS profileImage LIMIT 10;	Retrieve users who post questions containing the same tag of the questions asked given by a given user
MATCH (u3:User)<-[:FOLLOW]-(u2:User) <-[:FOLLOW]-(u:User displayName: "???") WHERE u3 <> u and u3 <> u2 RETURN distinct u3.displayName AS Username, u3.profileImage AS profileImage LIMIT 20;	Given a user Retrieve a list of "friends of friends", namely people that the user does not follow by are followed by the people the user itself is following

As depicted in the above tables, most queries requires the displayName of the user. In order to avoid that this parameter could be a bottleneck became a bottleneck for the entire graph a FULL_TEXT index has been added to the db. The Benchmarks show that such index could increase the search performance every time, at least, of a factor of 10.

3.3 CrossDatabase Coherence

In our application there are some operations, like insert a new User or eliminate a Post, that have to access to both Document and Graph databases. Of course we want that if an operation is performed on the first one, it would be performed also in the other one. To do this we created an ***error.log*** file where possible errors during an operation are stored. Specifically the operations Insert/Remove User, Insert/Remove Post, Insert/Remove Answer and Insert/Remove Vote need to check if the operation is correctly performed on the first database, if it is not an error is appended into the file. On the follow a schema of the process is shown; for brevity, we show just two examples because the procedure is the same for all cases.

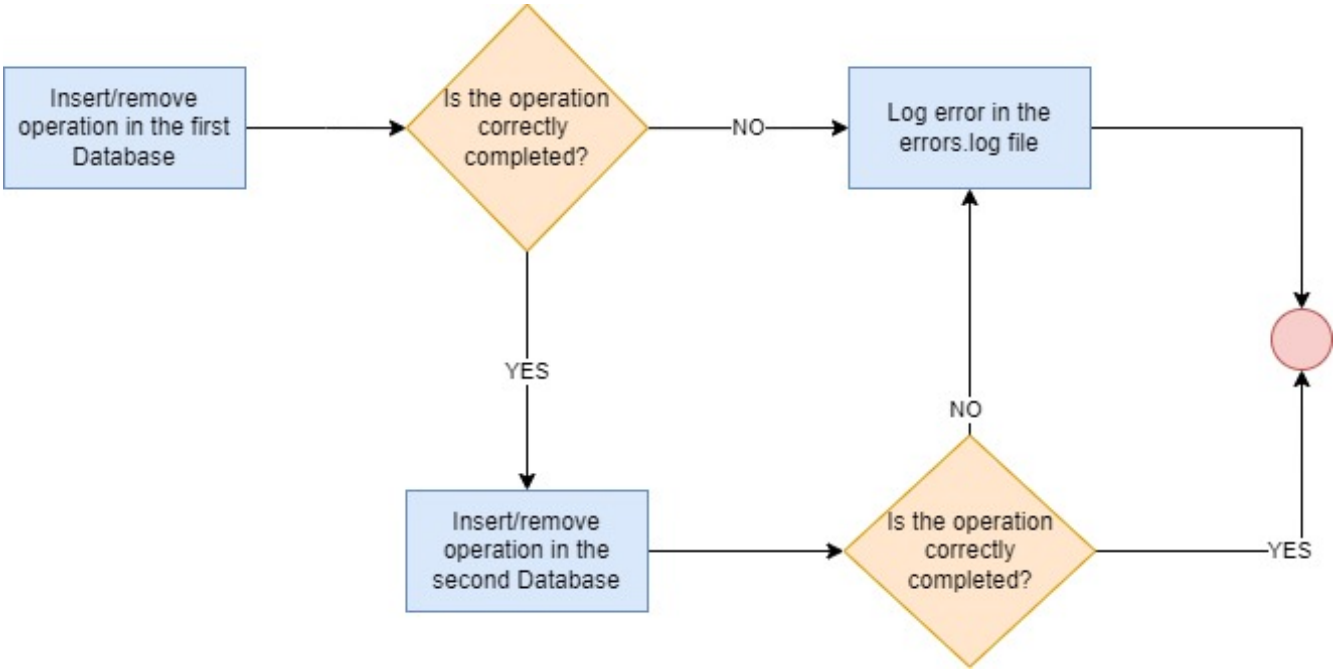


Figure 1: Cross-Database Coherence for an Insert/Remove operation

3.4 Cluster topology

The database cluster on which the server relies consists of 3 servers (available on the UNIPI network). An instance of MongoDB was deployed on every server (joined together in a replica set), meanwhile Neo4j was deployed on only one server, due to it being the Community Edition (thus not allowing the whole core/replica architecture).

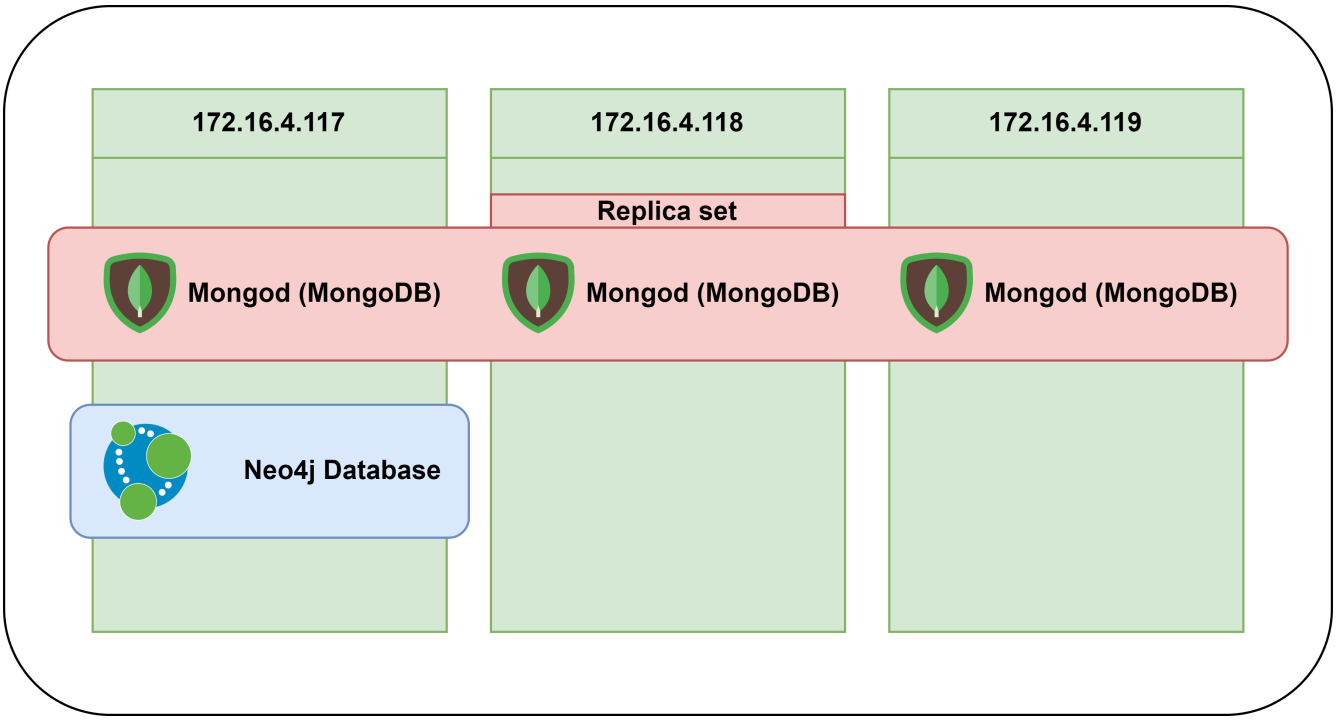


Figure 2: Cluster diagram

A further improvement on the availability could be achieved by deploying the document database on a sharded architecture. A possible sharding key capable of distribute in a relatively could be the _id field inserted by the DMBS itself. This field, in fact, is computed by means of an hash algorithm on the document itself. Hash algorithms are able per design to produce random values uniformly distributed on a quite wide spectrum of values(given by the digest length itself).

3.5 Server Package

The Server package contains a total number of seven Java classes; these classes are in charge to run the server, to handle the requests coming from the clients and to communicate with both document and graph databases. on the follow these classes are described more in detail.

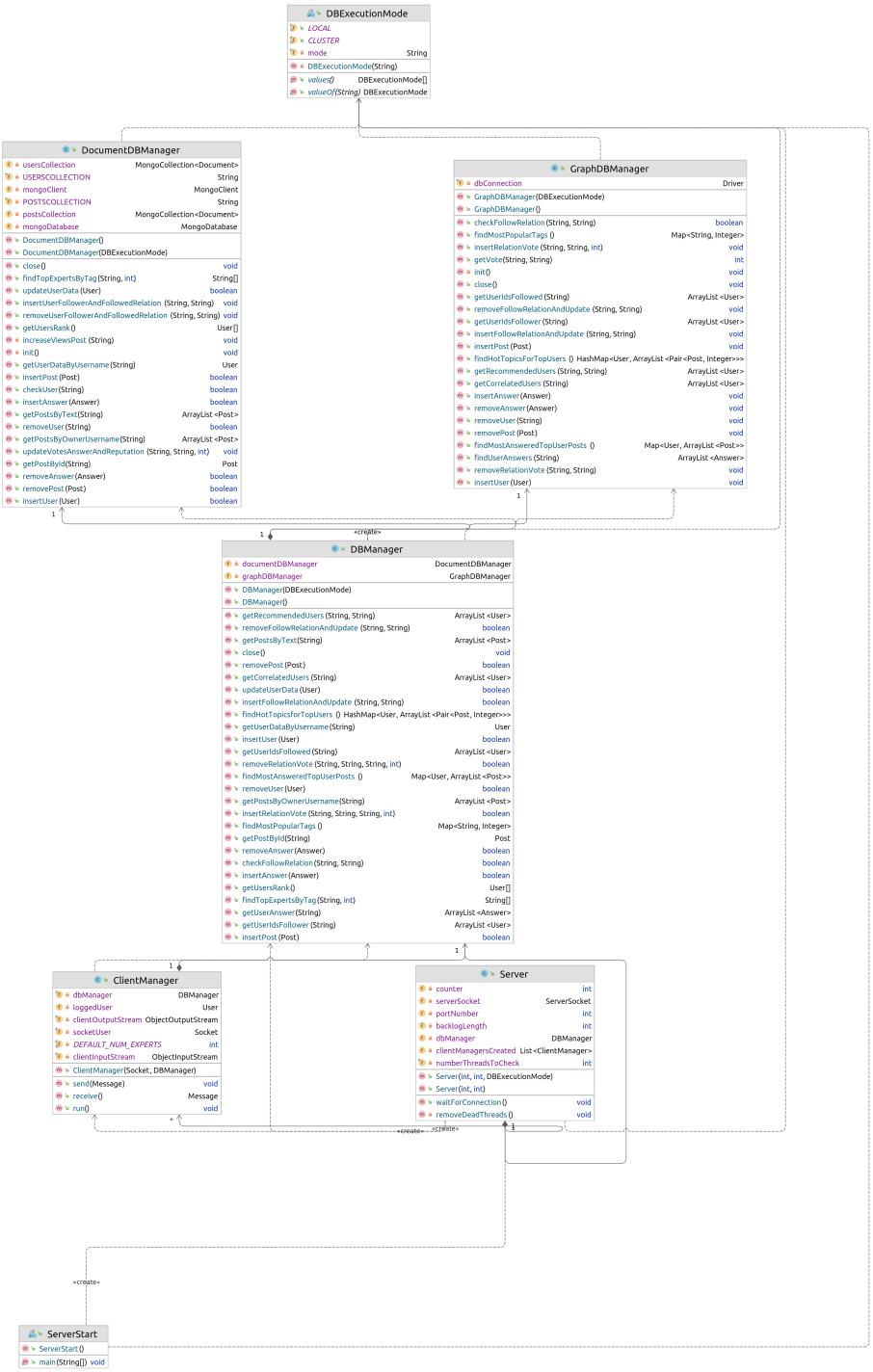


Figure 3: server package UML diagram

3.5.1 Server

The server class contains a constructor which take three arguments: the *port number* where the server listen for a client connection, the *backlog length* and the *execution mode* of databases, meaning local or cluster mode. The *waitForConnection* method is used to listen and accept a new connection with a client. Everytime a new connection is accepted, a new *ClientManager* instance is created to handle messages from the connected client.

3.5.2 ServerStart

This class contains the *main* method, so it is the class that actually run the server. In the main method we create a Server object and we wait for the connection with clients.

3.5.3 ClientManager

This class is instantiated when a new connection is created between server and client. This class handle the messages coming from the client: specifically a send and receive methods are used to send sand back and receive messages; these messages are then opened and forwarded to right query to be executed. The results of query are then sent back to the client.

3.5.4 DBExecutionMode

This class is a simple enumaration class to specify the methodology of databases execution between local and cluster.

3.5.5 DBManager

This class is supposed to communicate with the *ClientManager*, receiving the request to be forwarded to the databases, and on the other side with Graph and Document databases. So the DBManager forward to the right database (or to both of them) driver the client request. The constructor of this class create an instance of both Graph and Document database drivers.

3.5.6 DocumentDBManager

This class is the Java driver class to make queryes to the Document database, specifically to MongoDB. The *DBManager* forward the client requests to this class calling the method to run the right query. Results are then sent back to the *DBManager*. The constructor of this class take as parameter the execution mode, to run in local or into a cluster; moreover is opened the connection with the database, selecting the database and the collections needed.

GraphDBManager

This class is the Java driver class to make queries to the Graph database, specifically to Neo4j. The *DBManager* forward the client requests to this class calling the method to run the right query. Results are then sent back to the *DBManager*. The constructor of this class take as parameter the execution mode, to run in local or into a cluster; moreover is opened the connection with the database giving database name and password.

3.6 Messages

The *Messages* package contains the serializable classes used by the *ServerConnectionManager* and *ClientManager* to send respectively requests to the server and send back results; the objects to be sent are these messages. We created one abstract serializable class, *Message*, from which we extended all other classes; moreover we added two more abstract classes to better distinguish the tasks of the extended messages. The package contains also some enumeration classes. On the follow are described in detail all messages we created.

Figure 4: messages package UML diagram

3.6.1 Message

This is an abstract class and the serializable class, from which all other Message classes are extended. This class implements the method *getOpcode*, which return the Opcode, meaning the code to describe the message used by the *ServerConnectionManager* and the *ClientManager* to forward correctly the message.

MessageLogin

This message extends the *Message* abstract class and is used to send a login request to the server, sending the User data, specifically username and password.

MessageLogOut

This message extends the *Message* abstract class and is used the logout from the application, sending the username of the logged user to the server.

MessageSignUp

This message extends the *Message* abstract class and is used to perform the registration of an unregistered user. Specifically into this message all data inserted by the user in the registration form are sent to the server.

3.6.2 MessageCreateDelete

This is an abstract class, extended from the *Message* class. All messages which extends this class are messages used to send a create or delete request to the server. This message implements the *getOperation* method, which return the Operation to be performed (create, delete, check), and a *getObject* method, to get the sent object.

MessagePost

This message extends the *MessageCreateDelete* abstract class and is used to send a creation/delete request of an User to the Server.

MessageUser

This message extends the *MessageCreateDelete* abstract class and is used to send a creation/delete request of a Post to the Server.

MessageAnswer

This message extends the *MessageCreateDelete* abstract class and is used to send a creation/delete request of an Answer to the Server.

MessageFollow

This message extends the *MessageCreateDelete* abstract class and is used to send a creation/delete/check request of a Follow relationship to the Server.

MessageVote

This message extends the *MessageCreateDelete* abstract class and is used to send a creation/delete request of a Vote relationship to the Server.

3.6.3 MessageReadObjectQuery

This is an abstract class, extended from the *Message* class. All messages which extends this class are messages used to get some content from the Server, such as User data or Posts list.This class implements the *getObject* method, used to get the objects sent to and from the Server.

MessageGetAnswers

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get all the Answers of the specified User.

MessageGetPostsByParameter

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get a list of Posts given a specific *Parameter*, chosen between Username, Id and Text.

MessageGetUserData

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get all the data of a specific User. To get these data the username of the User is given.

MessageGetPostData

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get all the data of a specific Post. To get these data the ID of the Post is given.

MessageGetCorrelatedUsers

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users given a specific username. Specifically the class contains an ArrayList of Users to contain the response of the Server.

MessageGetRecommendedUsers

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users given a specific tag. Specifically the class contains an ArrayList of Users to contain the response of the Server.

MessageGetExpertsByTag

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users. Specifically the class contains an Array of Users to contain the response of the Server and a tag String as parameter of the query.

MessageGetFollowData

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users, the followers or the Followed of a specific user. Specifically the class contains an ArrayList of Users to contain the response of the Server and a tag String as parameter of the query, an username String as parameter for the query and a Boolean to choose between Follower or Followed.

MessageGetTopUsersPosts

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users and relative Posts. Specifically the class contains a Map of Users and Posts to store the Server response.

MessageAnalyticHotTopics

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users and some data of relative Posts. Specifically the class contains a Map of Users and an ArrayList of Pair (Post, Integer) to store the Server response.

MessageAnalyticMPTags

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of tags and the count of them. Specifically the class contains a Map of String and Integer to store the Server response.

MessageAnalyticUserRanking

This message extends the *MessageReadObjectQuery* abstract class and is used to send a request to the server to get data of a list of Users. Specifically the class contains an Array of Users to store the Server response.

3.7 Client

This package contains all the classes of the client side, needed to create the interfaces and to receive the user requests and forward them to the Server. Specifically we have the *ClientInterface* class, the *ServerConnectionManager* class and a certain number of *Controllers*, described in the follow.

Figure 5: client package UML diagram

3.7.1 ClientInterface

This class extends the *Application* abstract class. It contains the *main* method an initialization method to initialize the interfaces and methods to handle them.

3.7.2 ServerConnectionManager

This class extends the *Thread* class. Is is supposed to create the connection with the Server side and to receive requests from the interfaces and forward them to the Server side. This class implements also a send and a receive methods to be able to communicate with the *ClientManager* on the Server side.

3.7.3 Controllers

These classes are utilized to control the interfaces we created. We have a controller for each interface, and each one of them contains methods that fill all the fields of the respective interface and also methods that triggers when the user interact with the interface, like selecting a Post or clicking a button.

4. User manual

4.1 Premise

The following section contains some procedures that users can follow in order to use correctly the client. All operations have been done with the following users:

Username	Password	Type
jMarcos	jMarcos	User
Armstrong	Armstrong	User
nuovoUtente	nuovoUtente	User
tizio123	tizio123	Admin

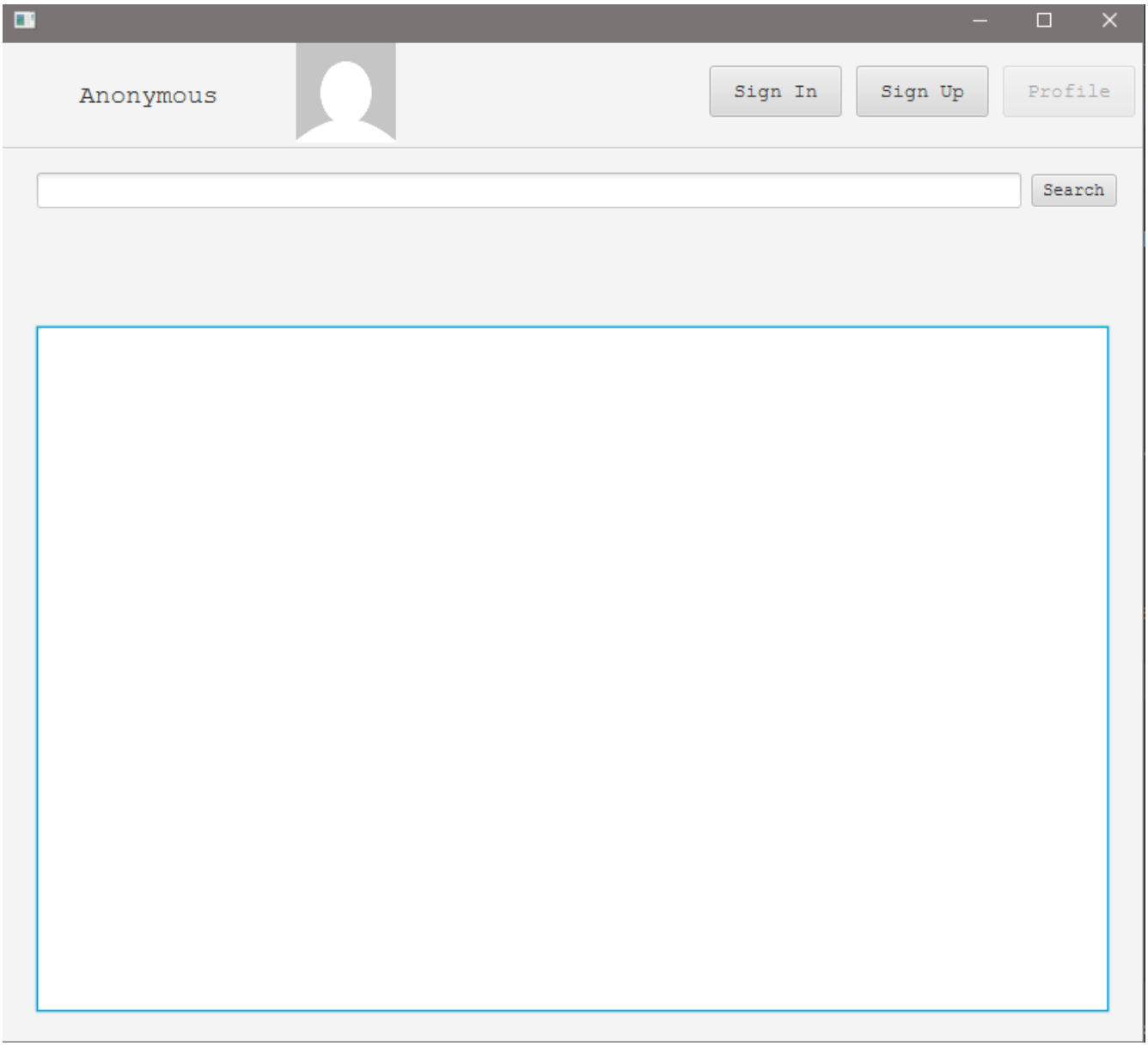


Figure 6: Home screen

4.2 Register a new user

1. Starting from the home screen (Fig. 6), click the "Sign Up" button
2. Insert the required details and click the "Register" button (Fig. 7)
3. You'll be redirected to the new profile that was just created (Fig. 8).

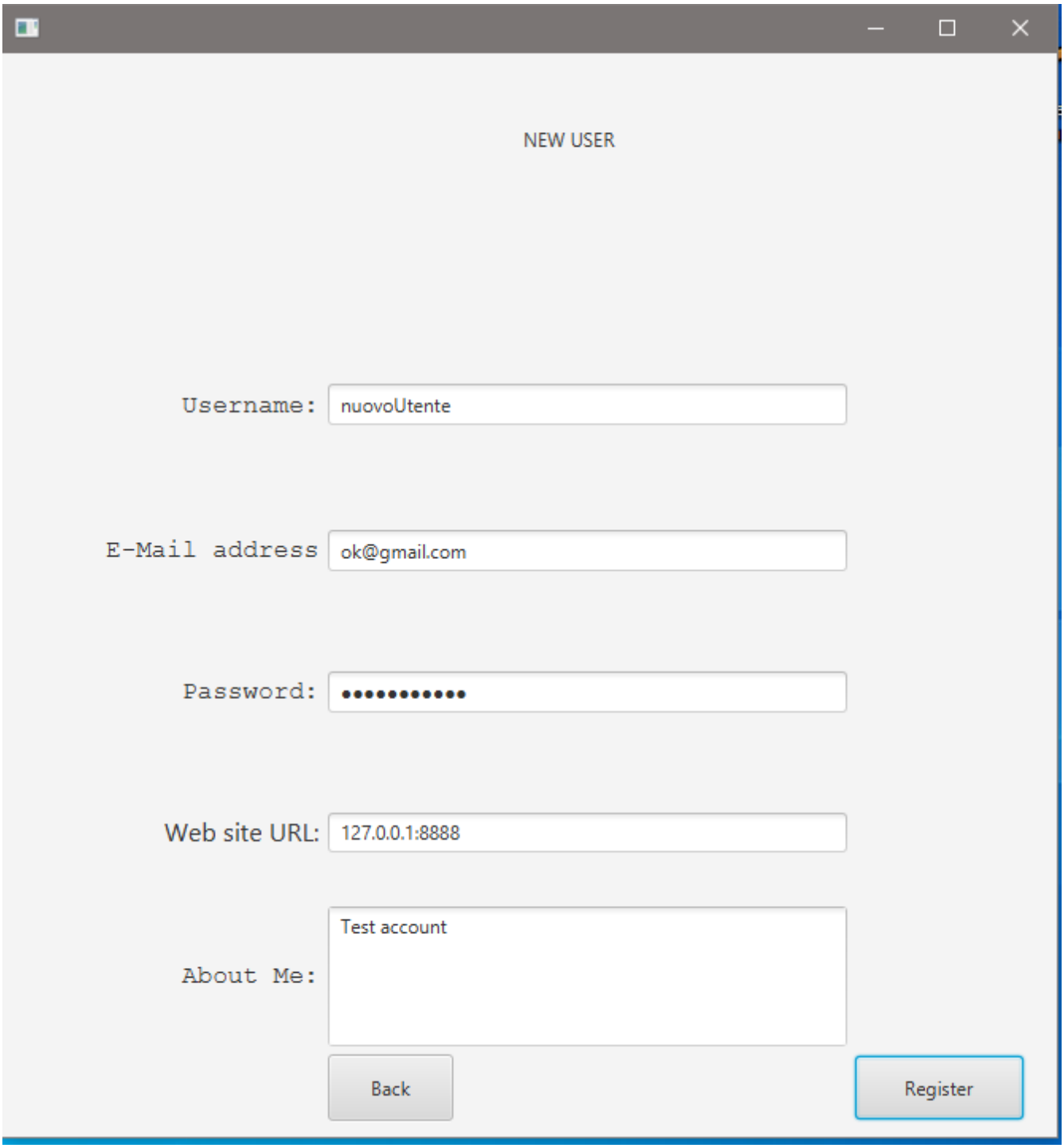


Figure 7: Register page

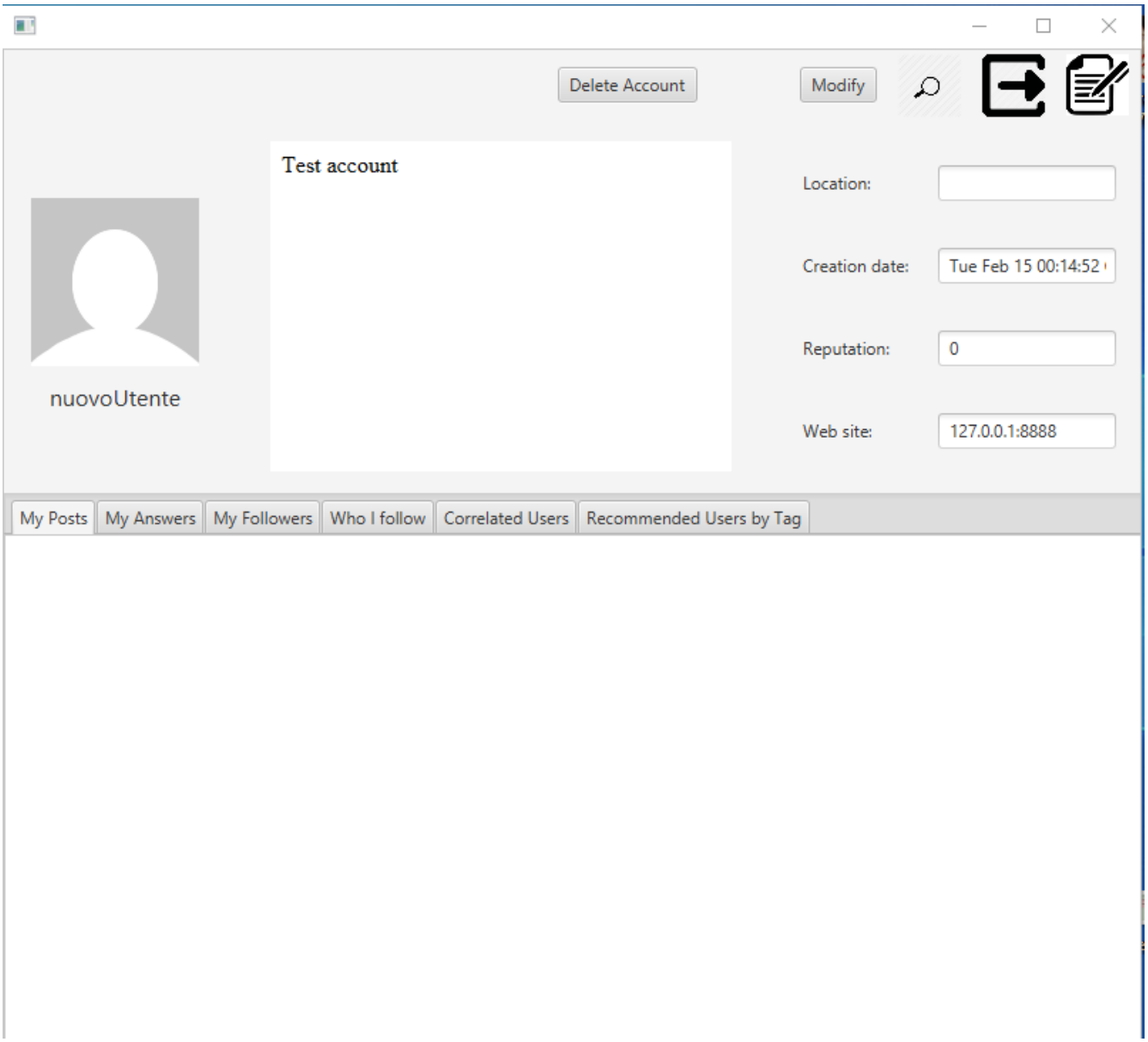


Figure 8: New user profile

4.3 Login

1. Starting from the home screen (Fig. 6), click the "Sign In" button
2. Insert the username and the password, and then click the "Sign In" button (Fig. 9)
3. You'll be redirected to the user profile (Fig. 10).

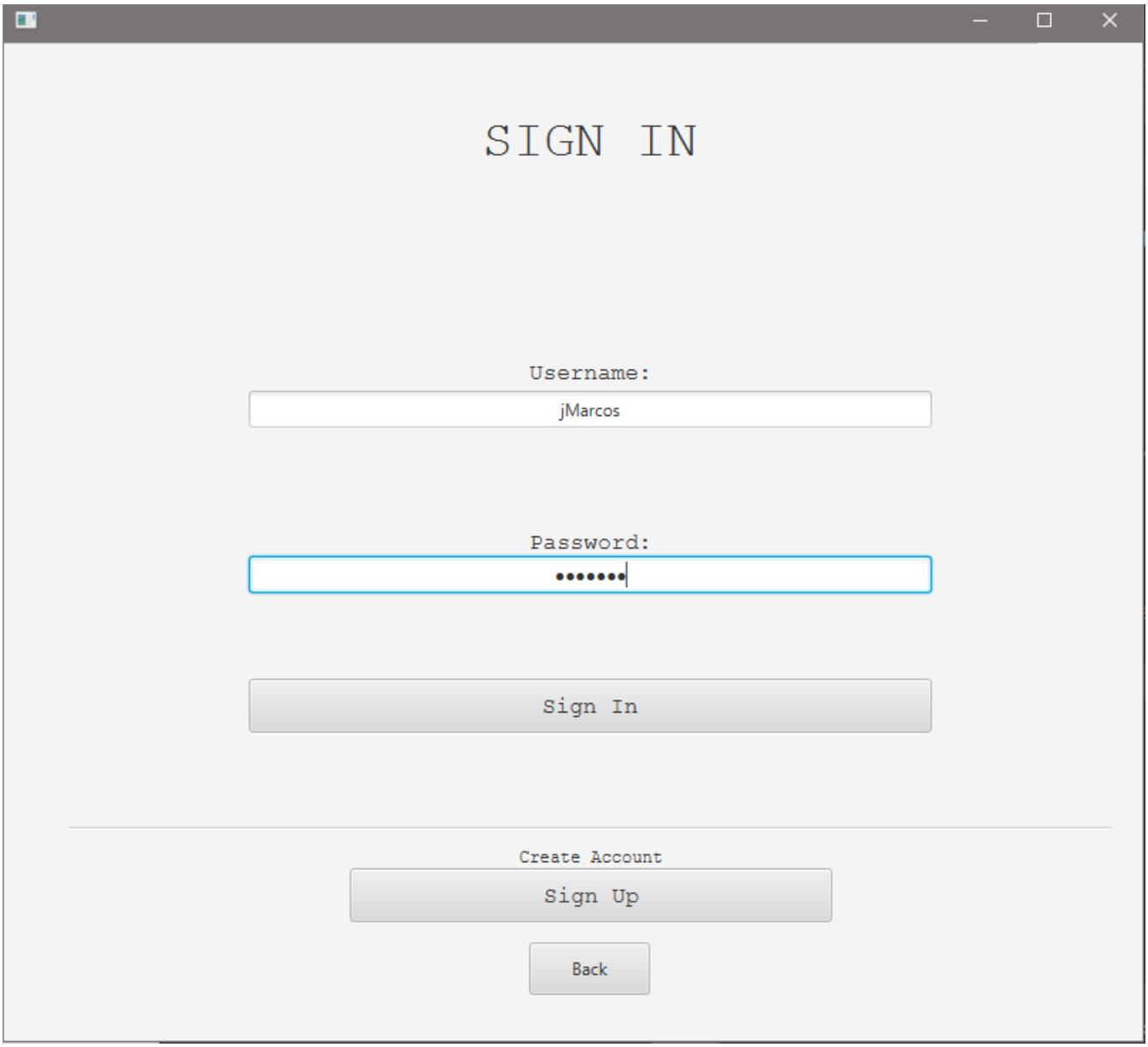


Figure 9: Login page

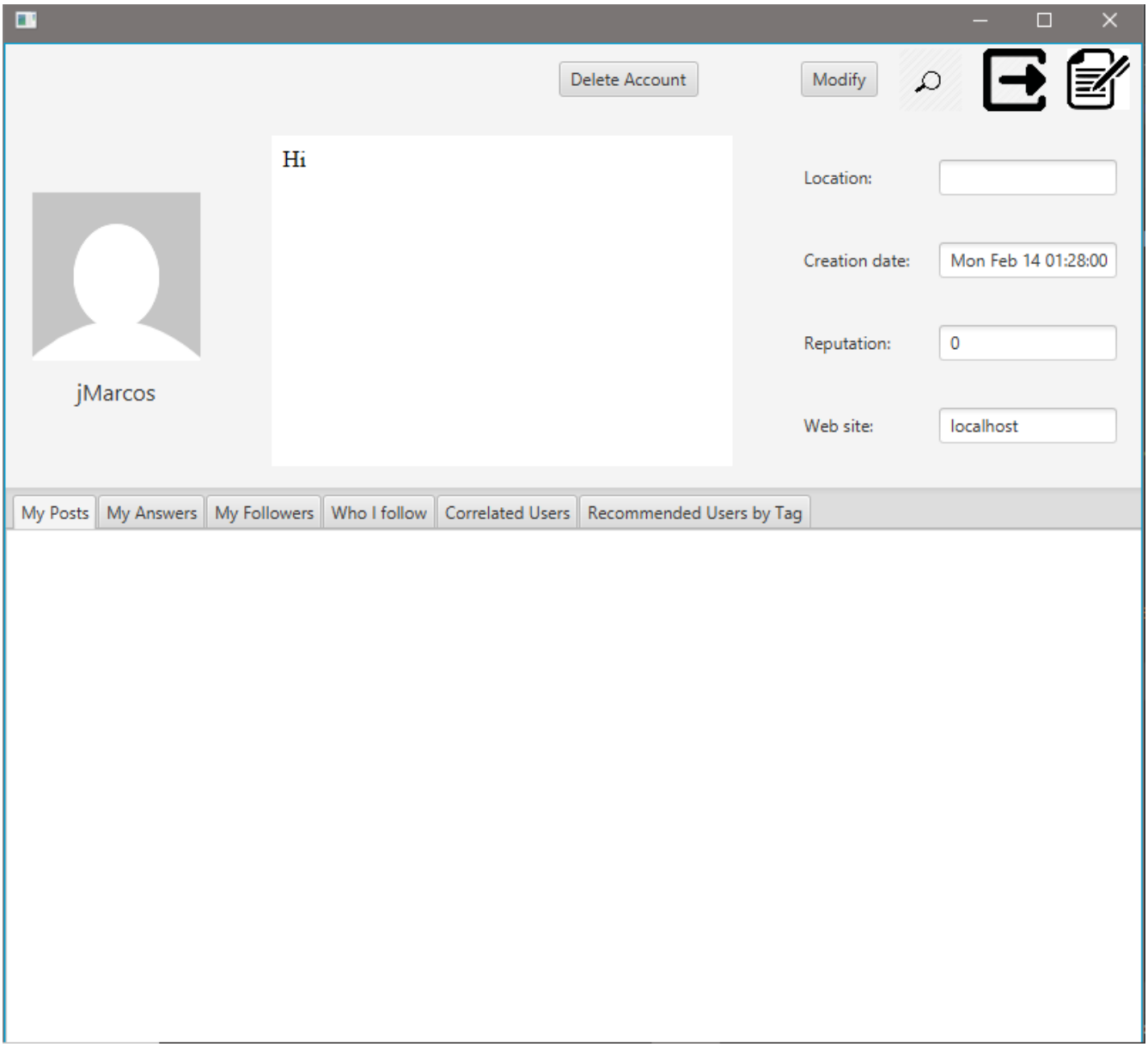


Figure 10: User profile

4.4 Logout

1. Starting from your user profile (Fig. 11), click the icon on top with an arrow going on the right
2. You'll be redirected to the home screen (Fig. 6).

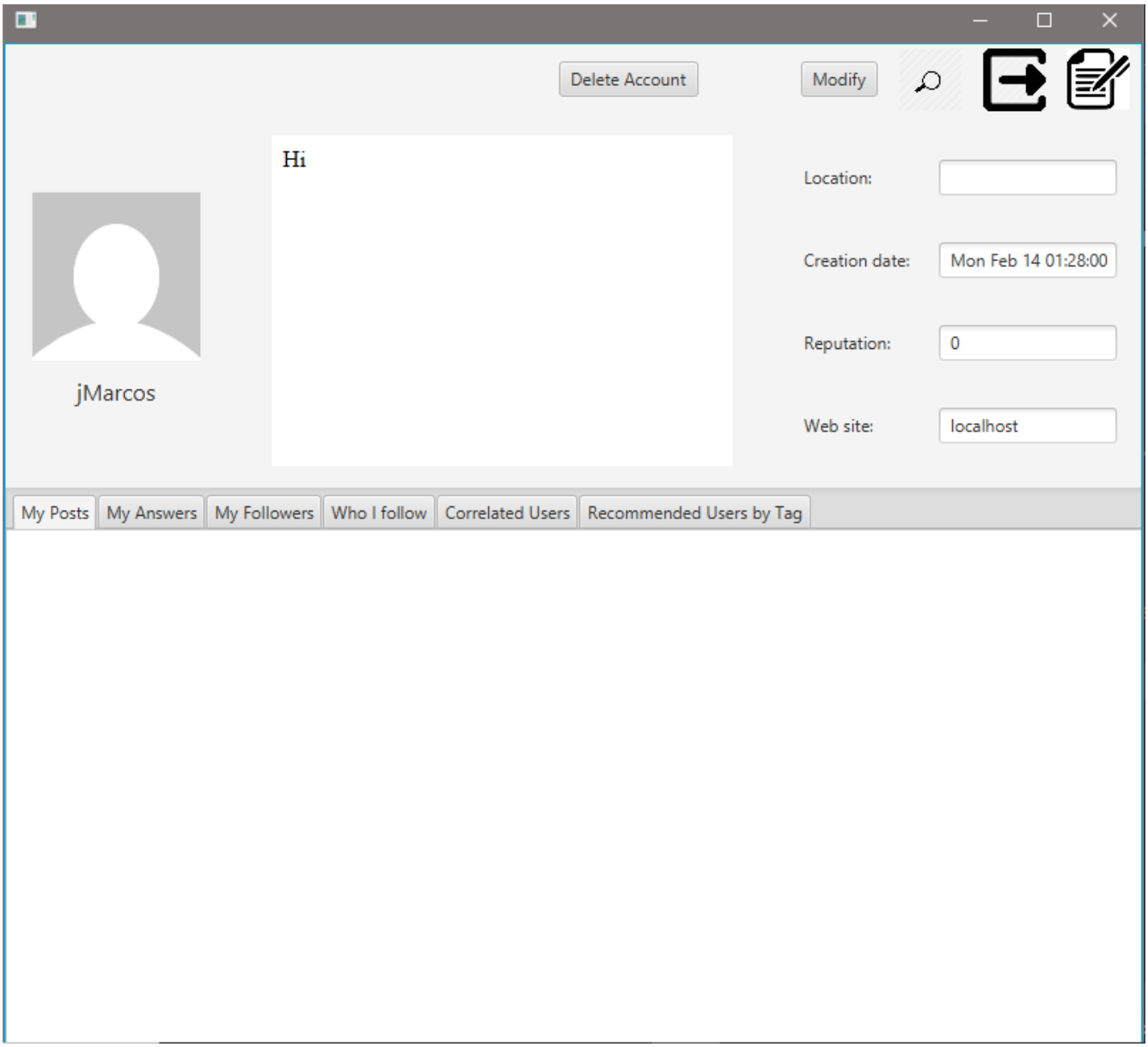


Figure 11: User profile

4.5 Write a new post

1. Starting from your user profile (Fig. 10), click the icon on the top-right (the document with the pen)
2. Insert the title, the tags used (separated by the ";" character), and the body (Fig. 12)
3. You'll be redirected to the user profile, and in the "My Posts" tab, there will be a reference to the newly created post (Fig. 13).

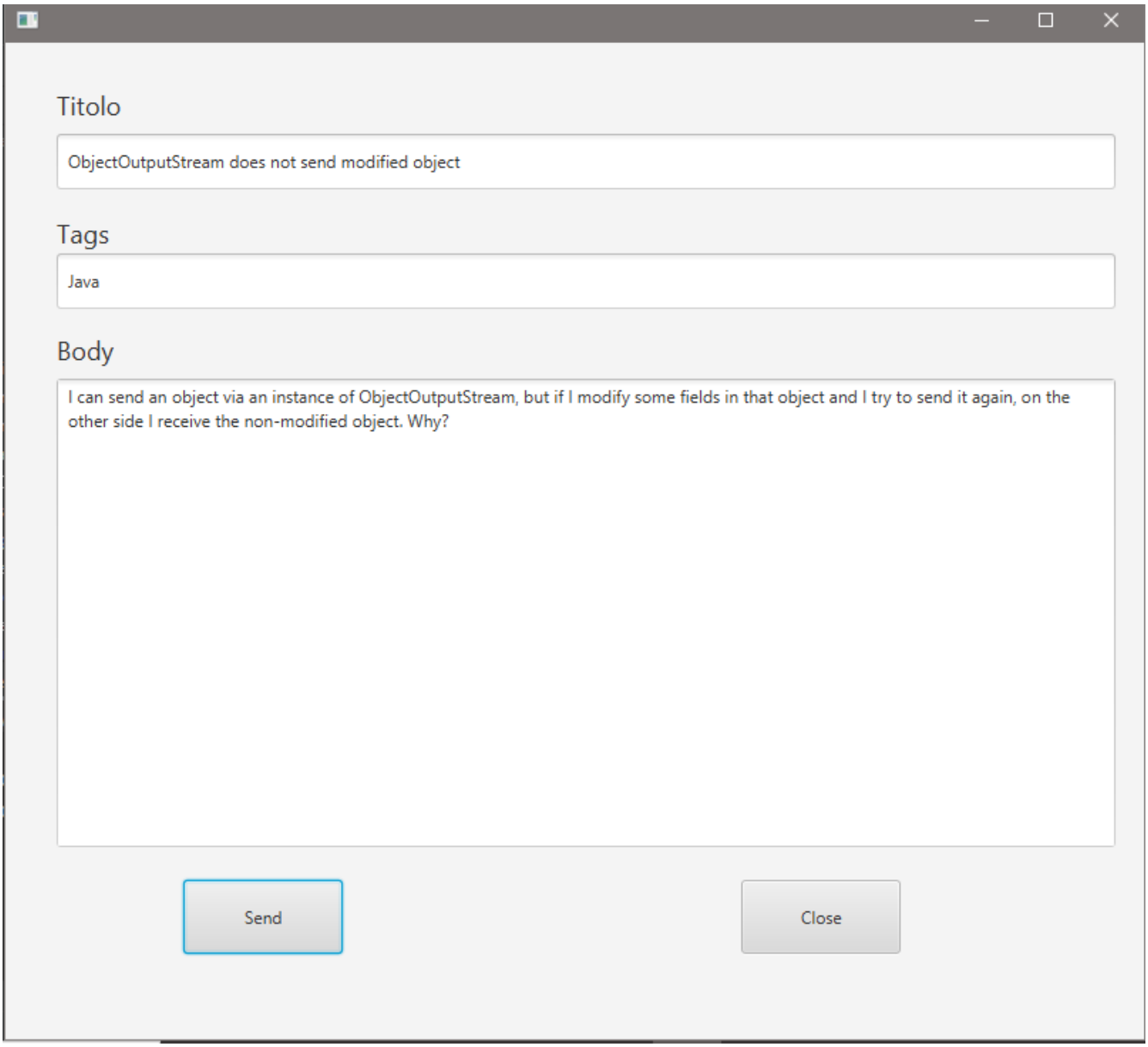


Figure 12: New post interface

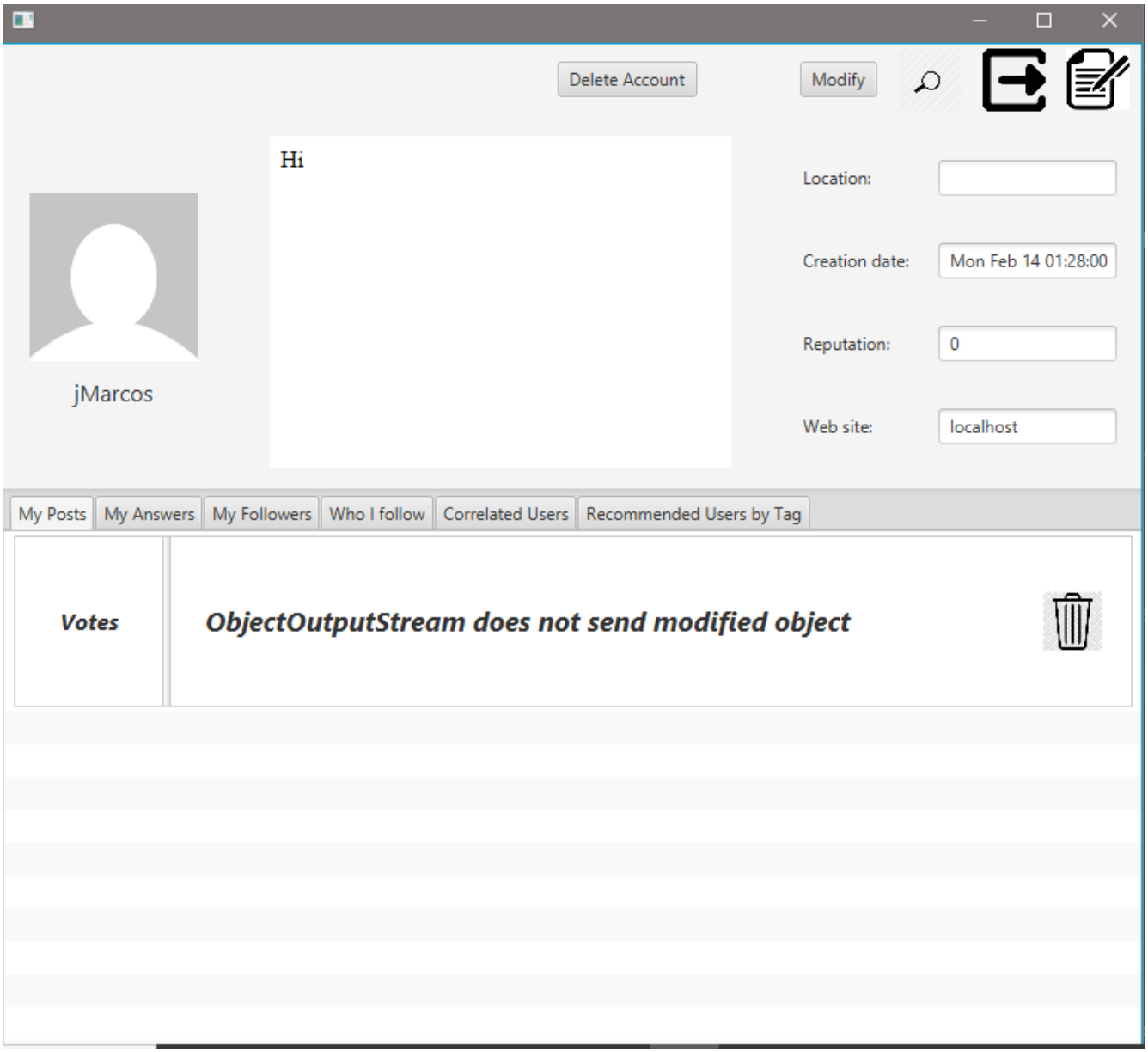


Figure 13: User profile after writing a new post

4.6 Find a post

1. Starting from your user profile (Fig. 13), click the magnifying glass icon on the top-right
2. In the search box, insert some keyword related to what you're looking for, and then click the "Search" button (Fig. 14)
3. If there are posts, they will appear on the big box below. Click on any post (Fig. 15)
4. Inside the post interface you can see the body, the (if present) answers, and a textbox for inserting your answer to the question (Fig. 16). You can also click the "Back" button to return to the post search page.

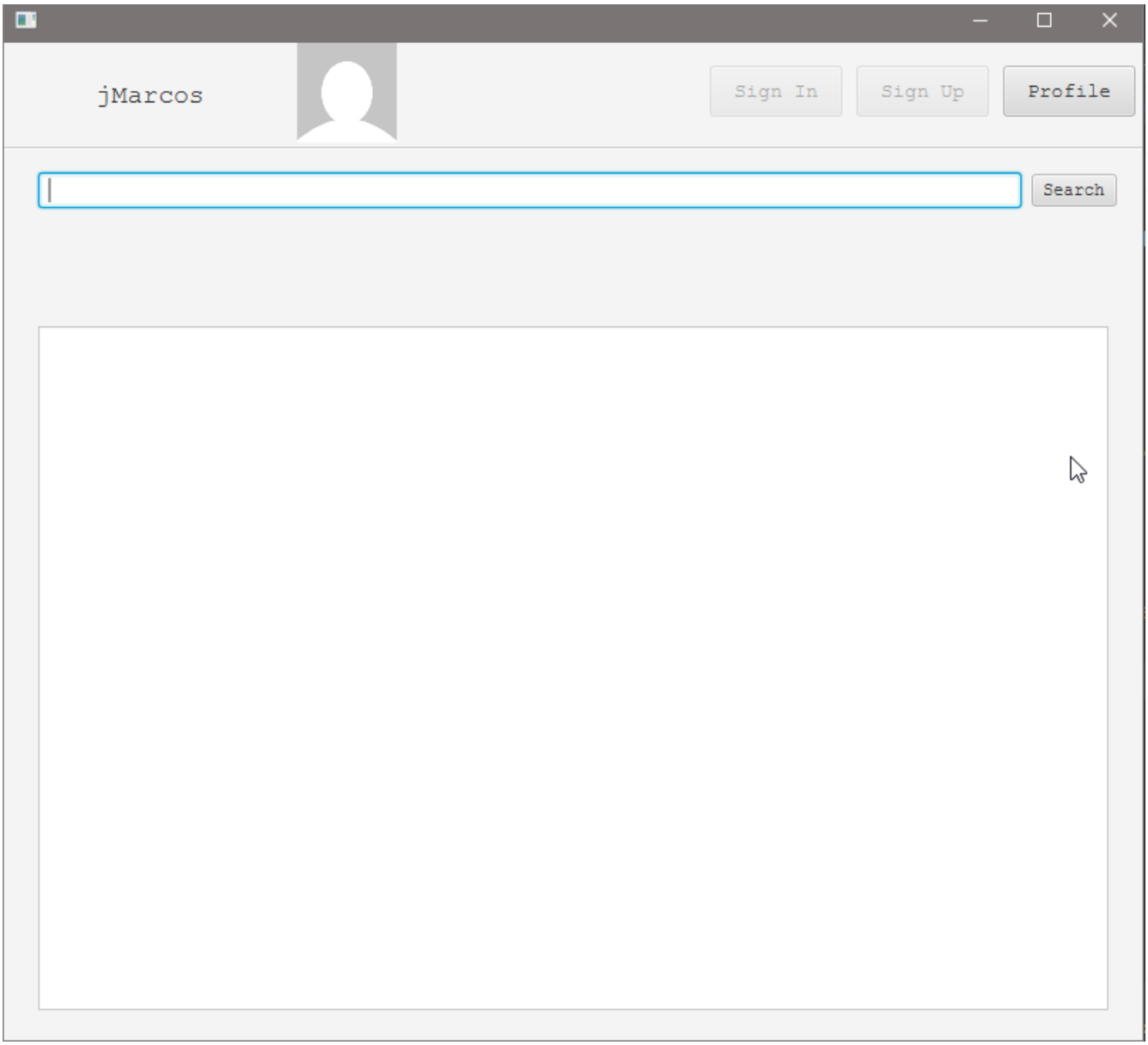


Figure 14: Post search interface

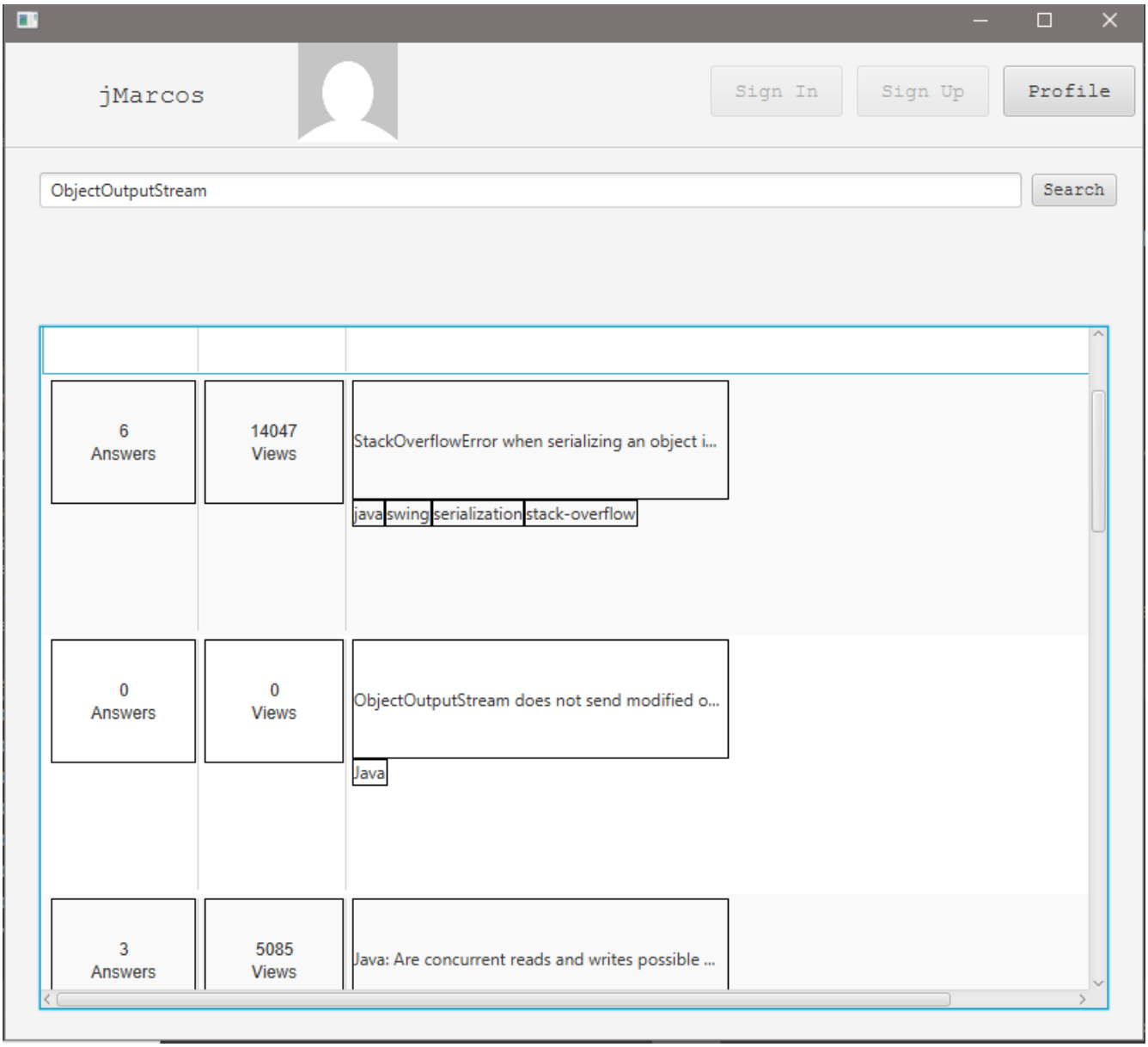


Figure 15: After clicking the search button

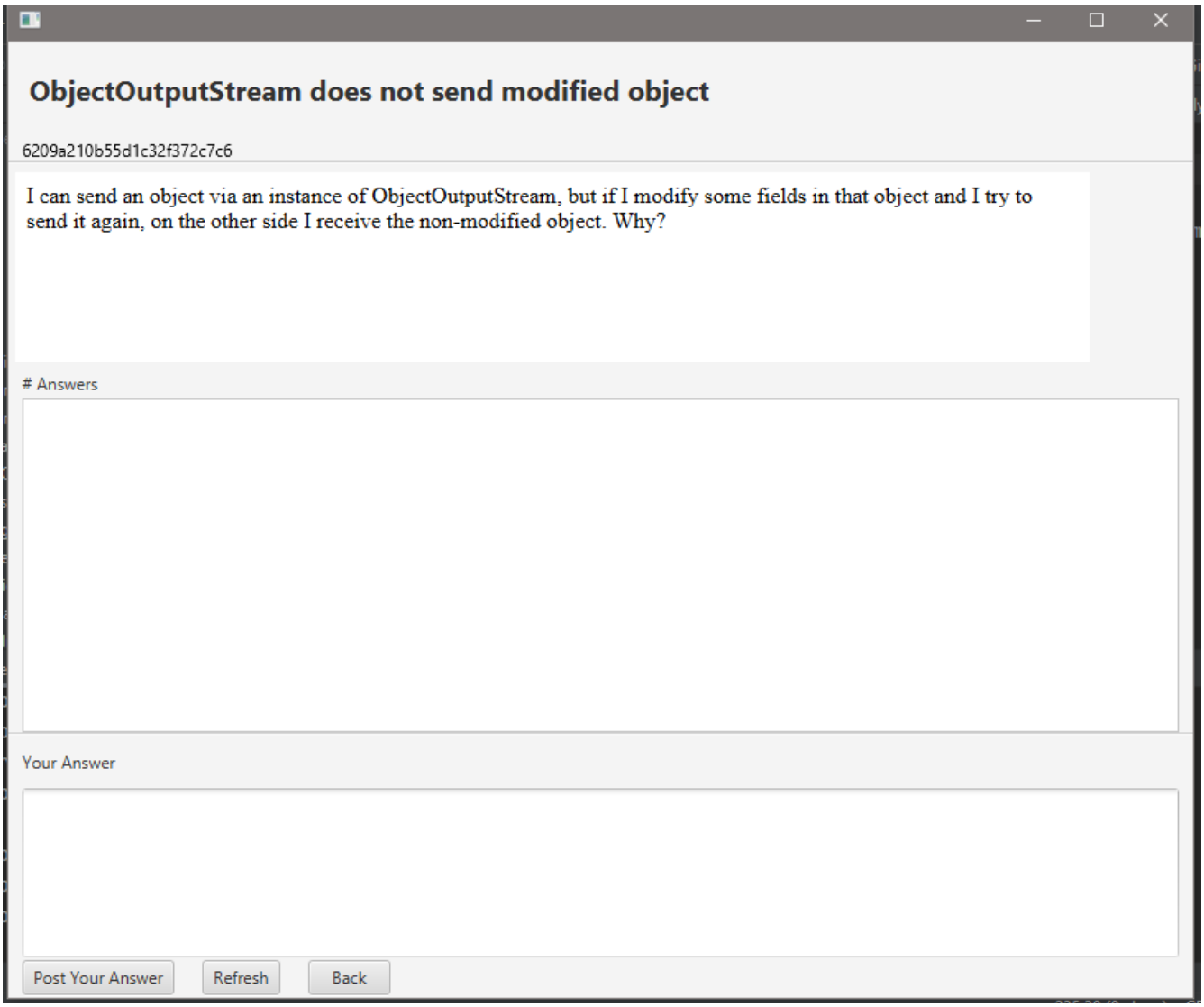


Figure 16: Post details page

4.7 Write an answer to a post

1. Starting from the user profile (Fig. 10), find a post (for example the one shown in Fig. 17)
2. In the "Your Answer" textbox, write your answer to the post, and then press the "Post your Answer" button (Fig. 17)
3. Press the "Refresh" button (or press "Back" and check again the post), and your answer will appear (Fig. 18).

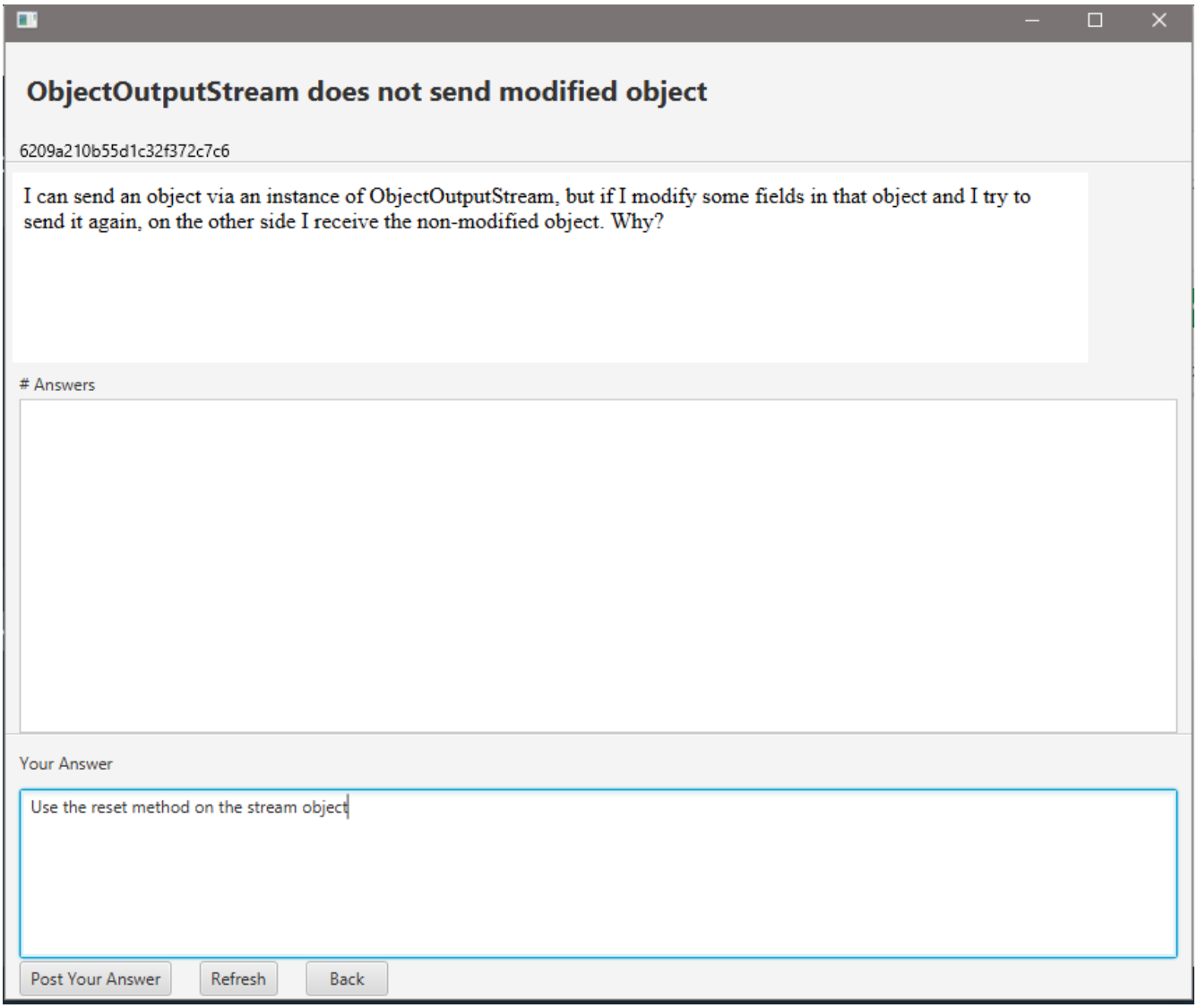


Figure 17: Before posting a new answer

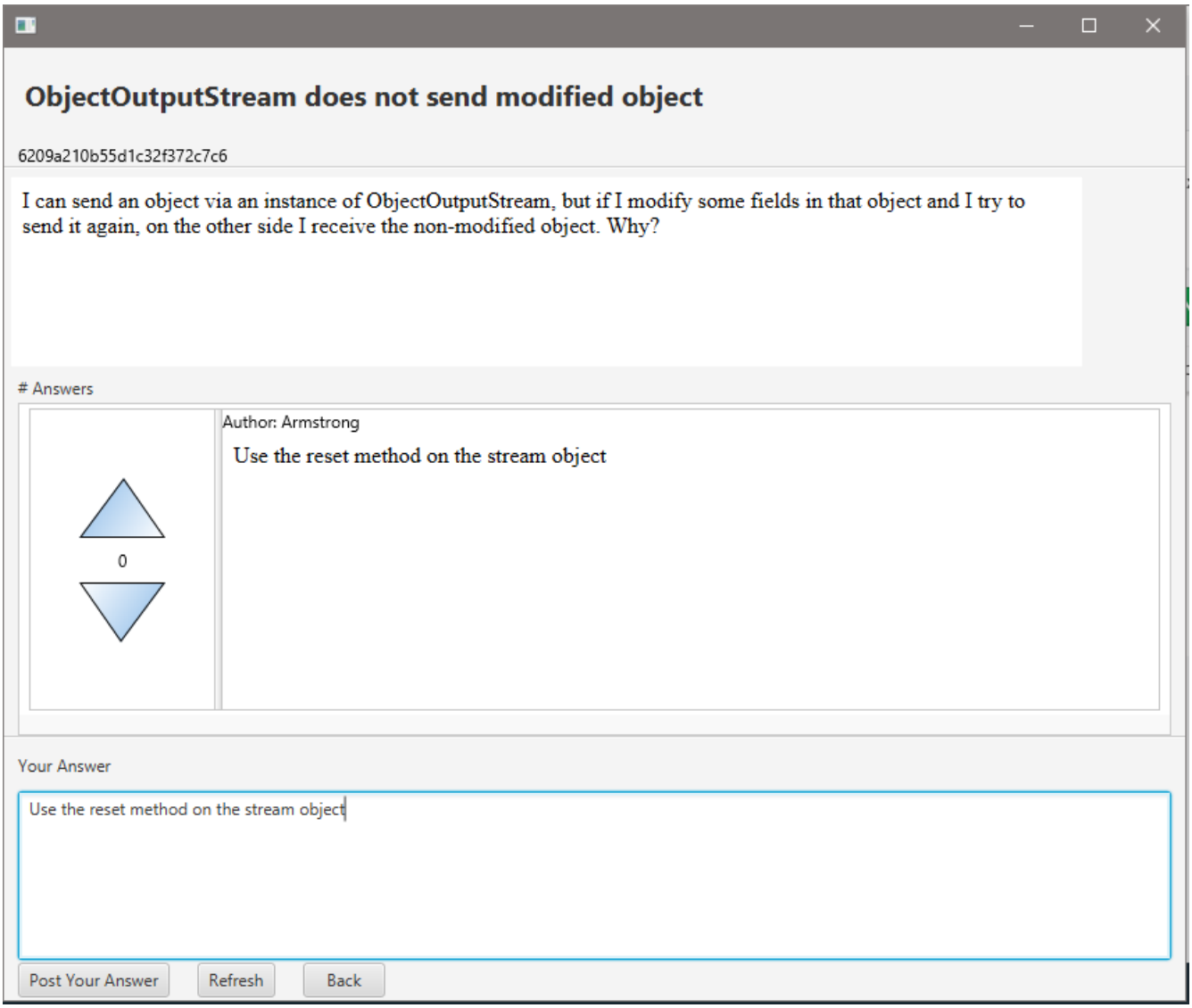


Figure 18: After posting an answer (and pressing Refresh)

4.8 Upvote or downvote an answer

1. Starting from the user profile (Fig. 10), find a post with at least an answer (for example the one shown in Fig. 19)
2. Supposing that you want to upvote an answer, click on the Up triangle on the left of the answer (Fig. 20), then click the "Refresh" button
3. Now the answer will show its new score (Fig. 21).

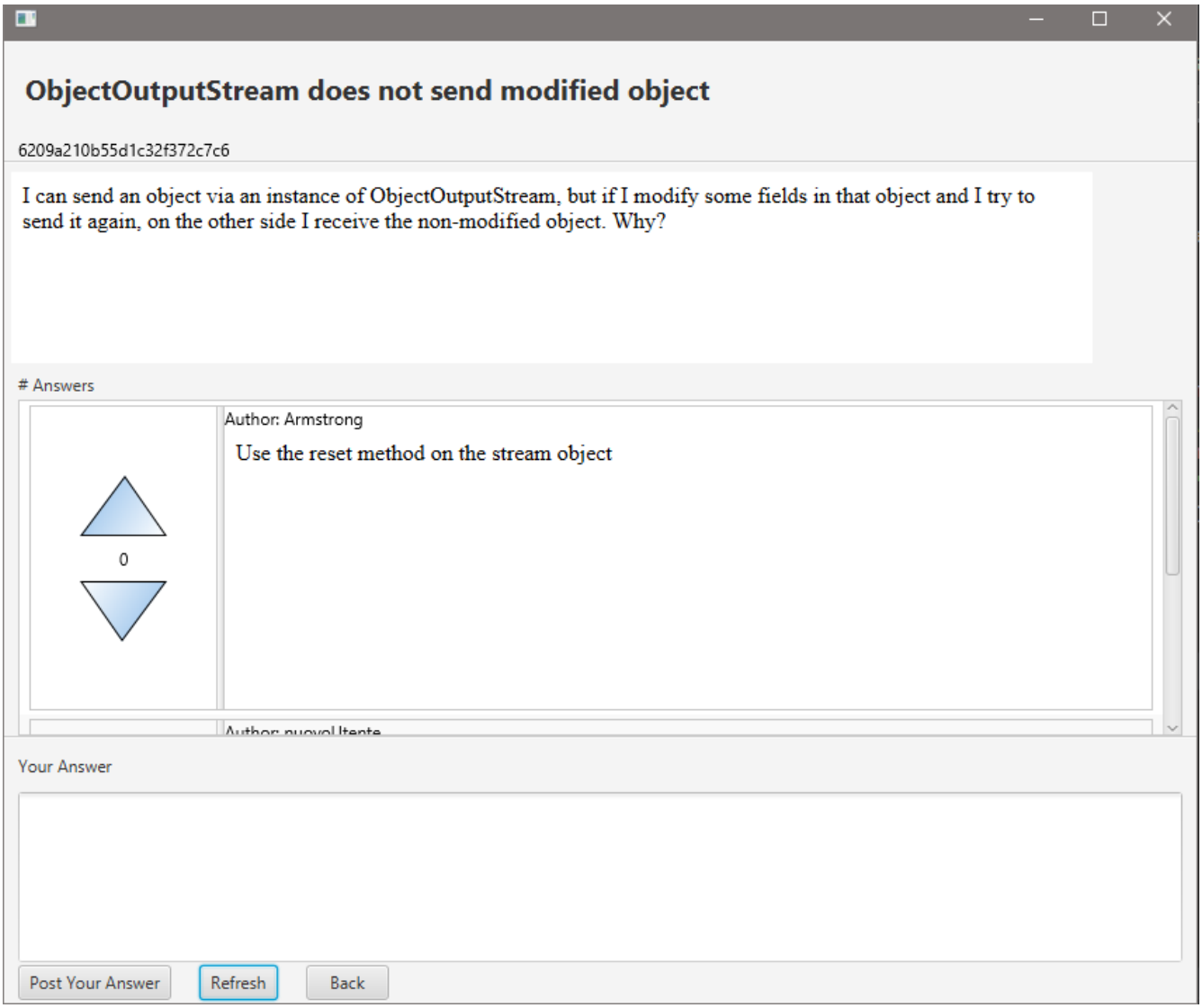


Figure 19: Post with at least an answer

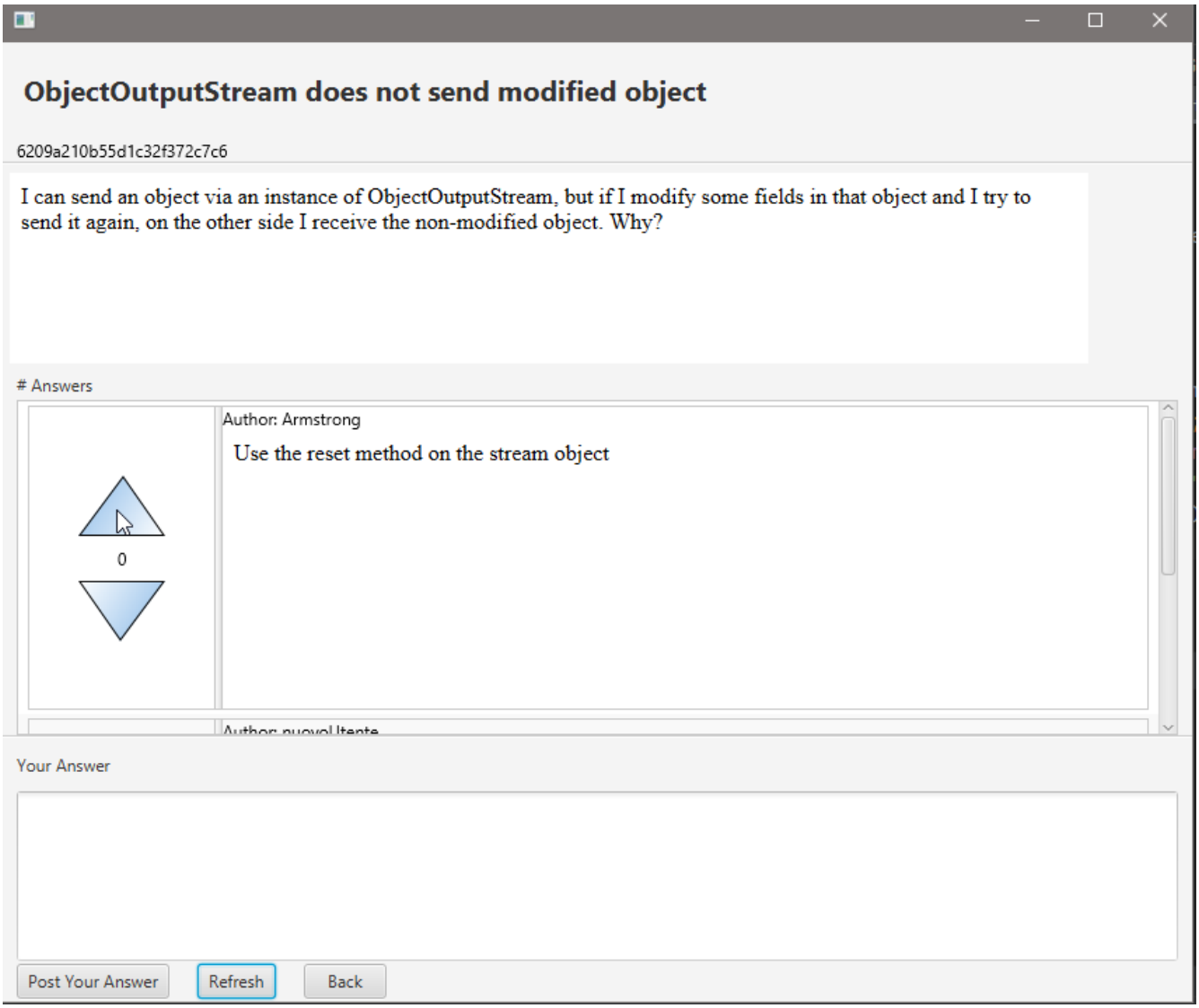


Figure 20: Upvoting an answer

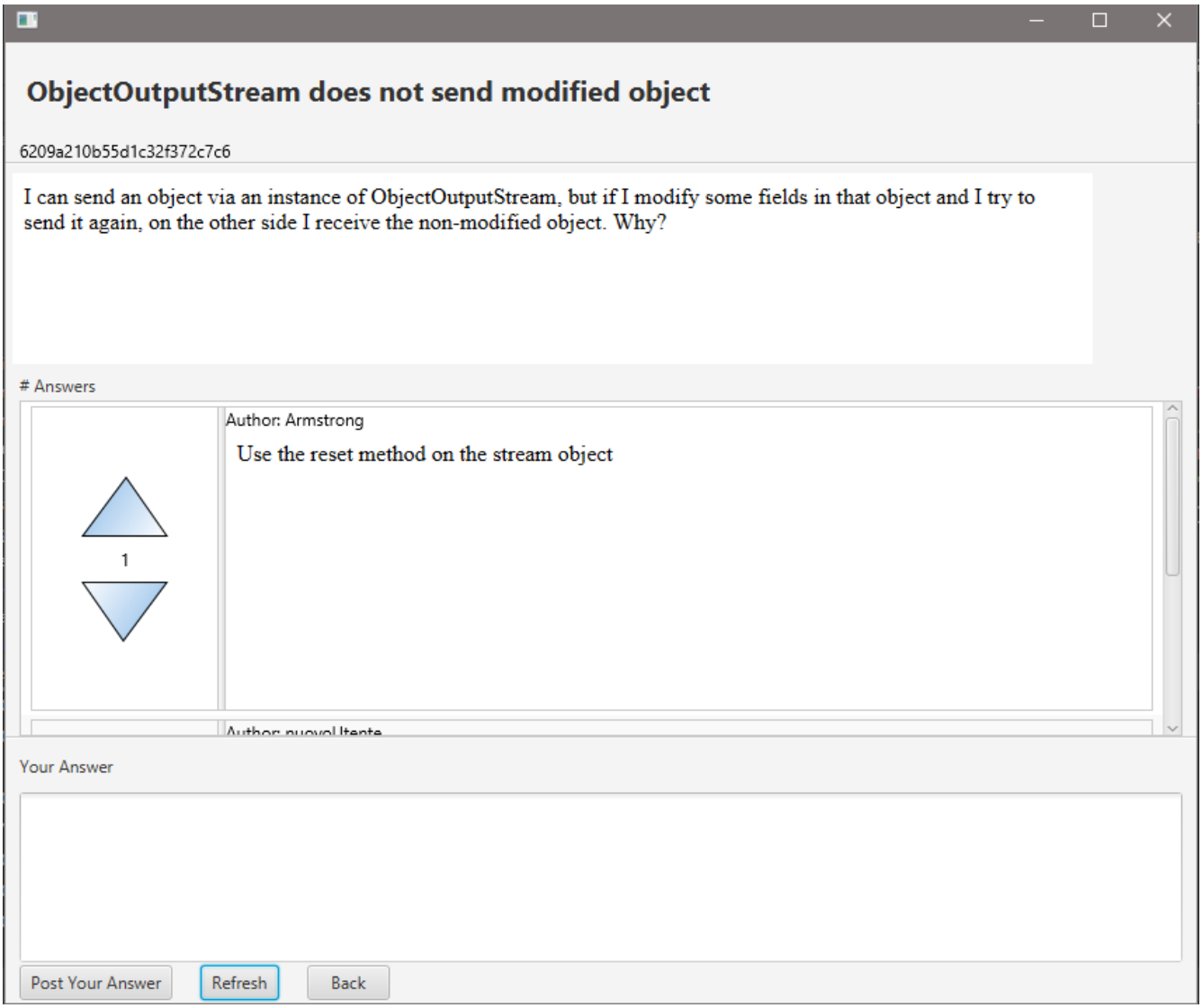


Figure 21: After the upvote (and Refresh)

4.9 Modify user details

1. Starting from the user profile (Fig. 10), click on the "Modify" button on the top (Fig. 22)
2. Currently it’s possible to change the location, the website, and the About me description (the big centered textbox). After changing their values, click on the "Save" button (Fig. 23)
3. Now the new values will appear in the profile (Fig. 24).

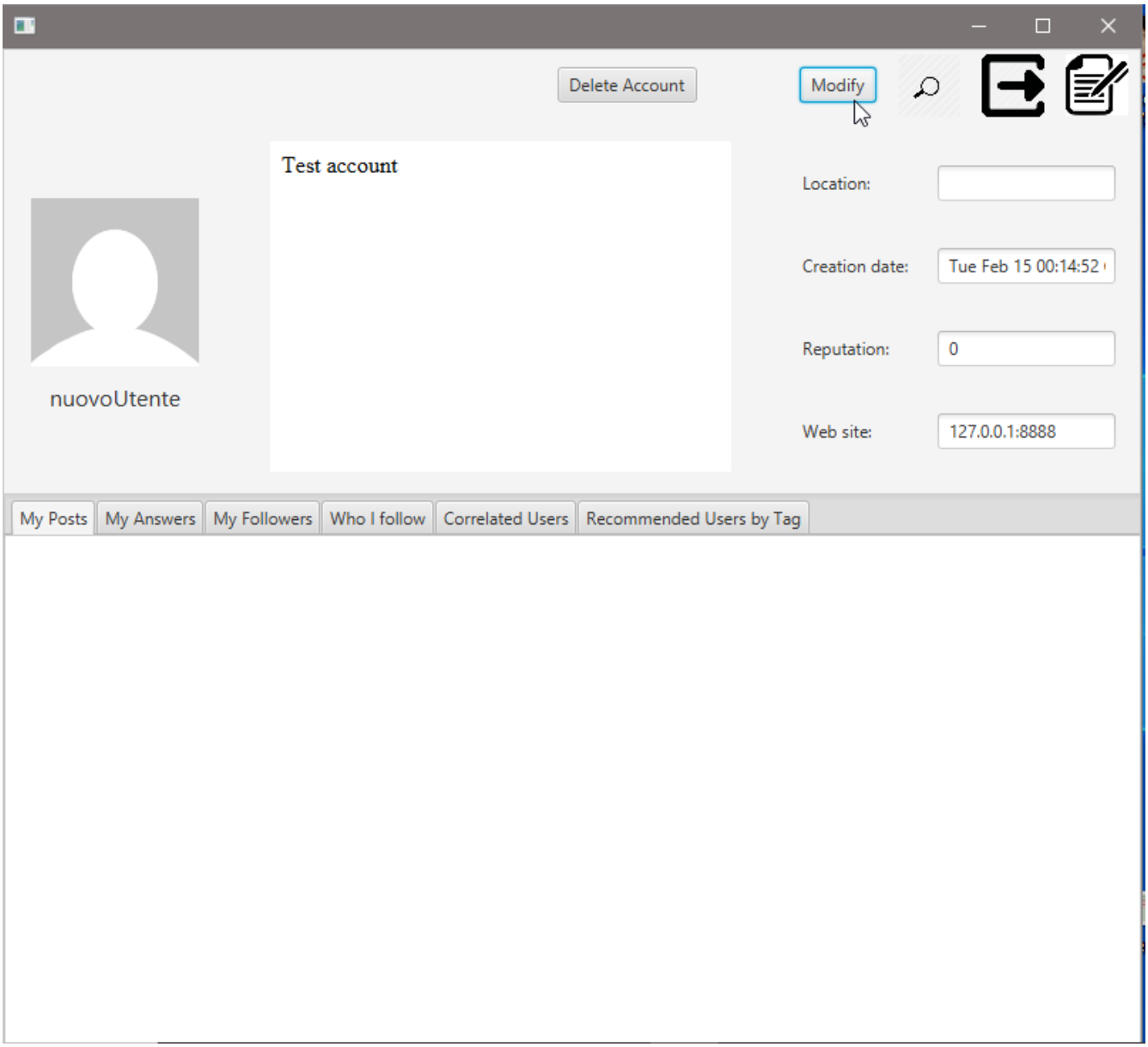


Figure 22: Before any changes

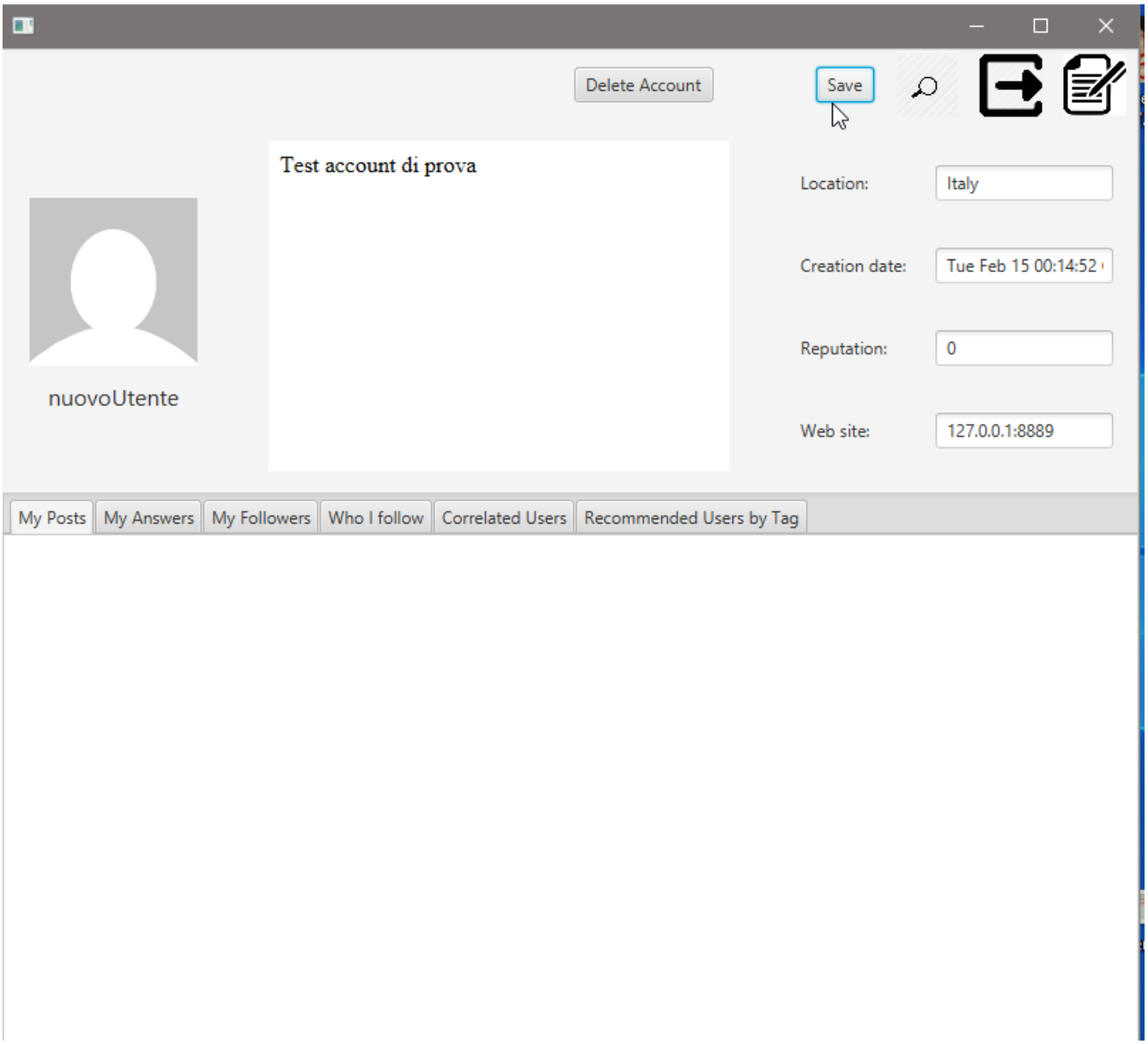


Figure 23: Modifying the user profile

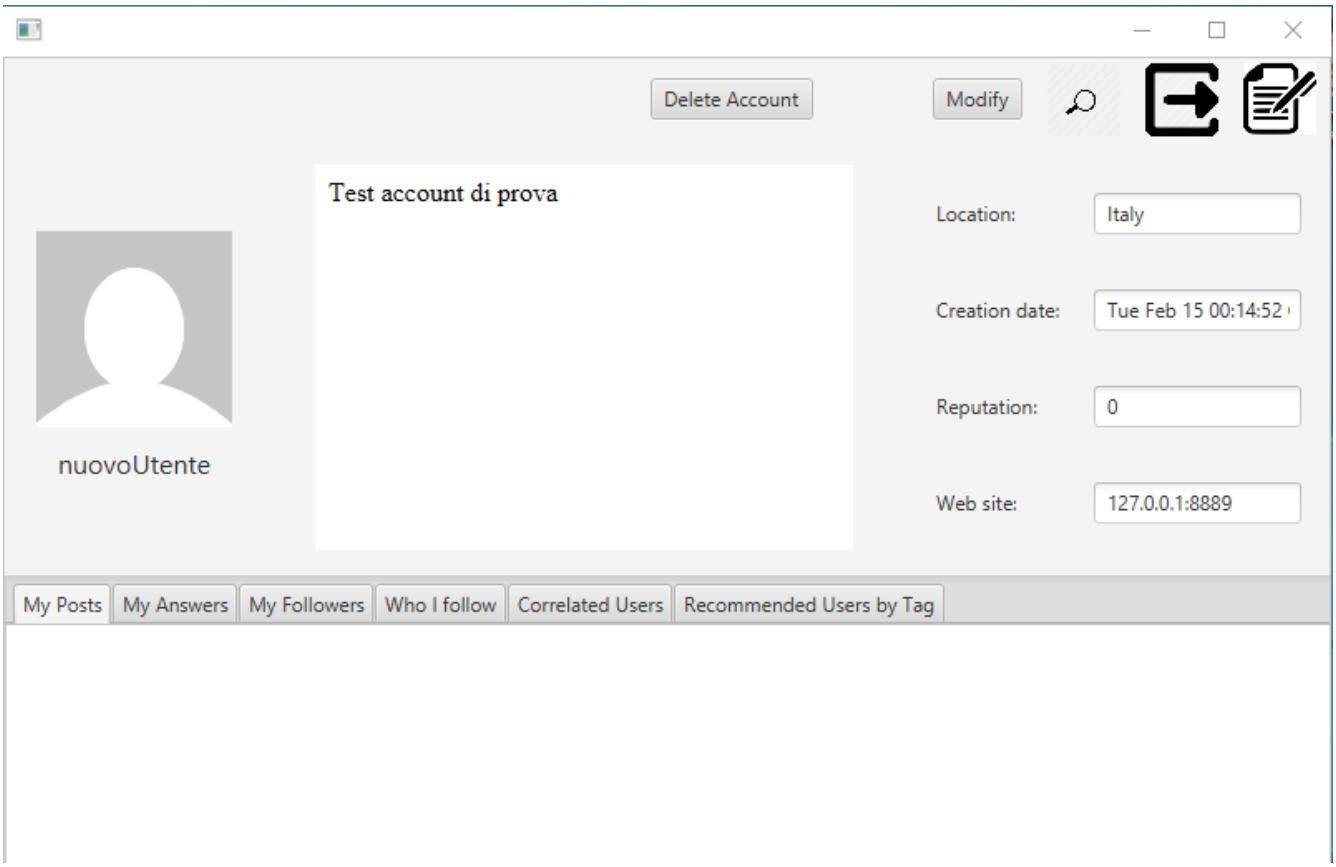


Figure 24: After saving the changes

4.10 Check user who wrote an answer

1. Starting from the user profile (Fig. 10), find a post with at least an answer (like in Fig. 25)
2. Right-click on the answer itself, and then click the "See answer writer profile" option (Fig. 26)
3. Details about the user who wrote the answer will now appear on the screen (Fig. 27).

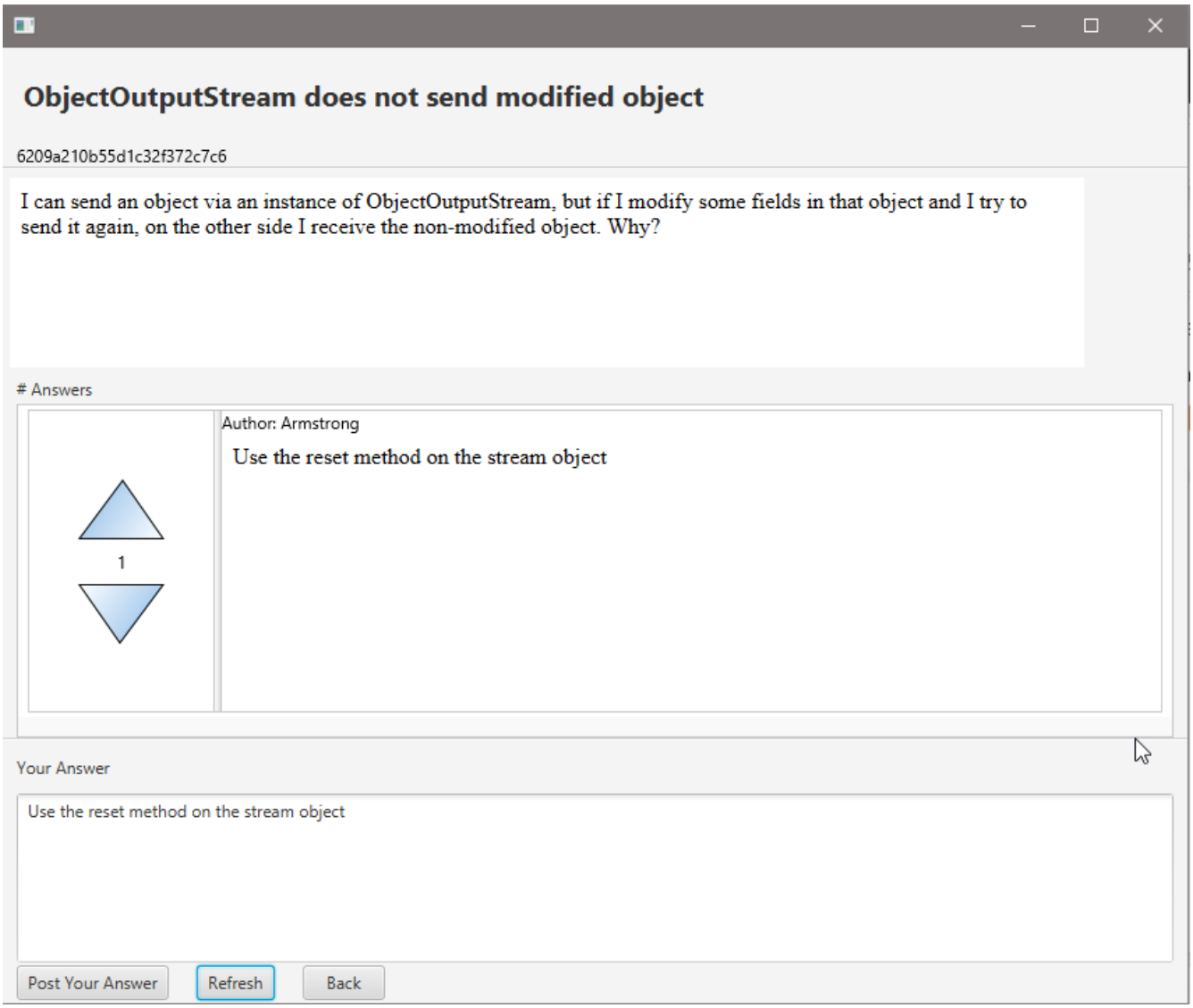


Figure 25: Post with at least an answer

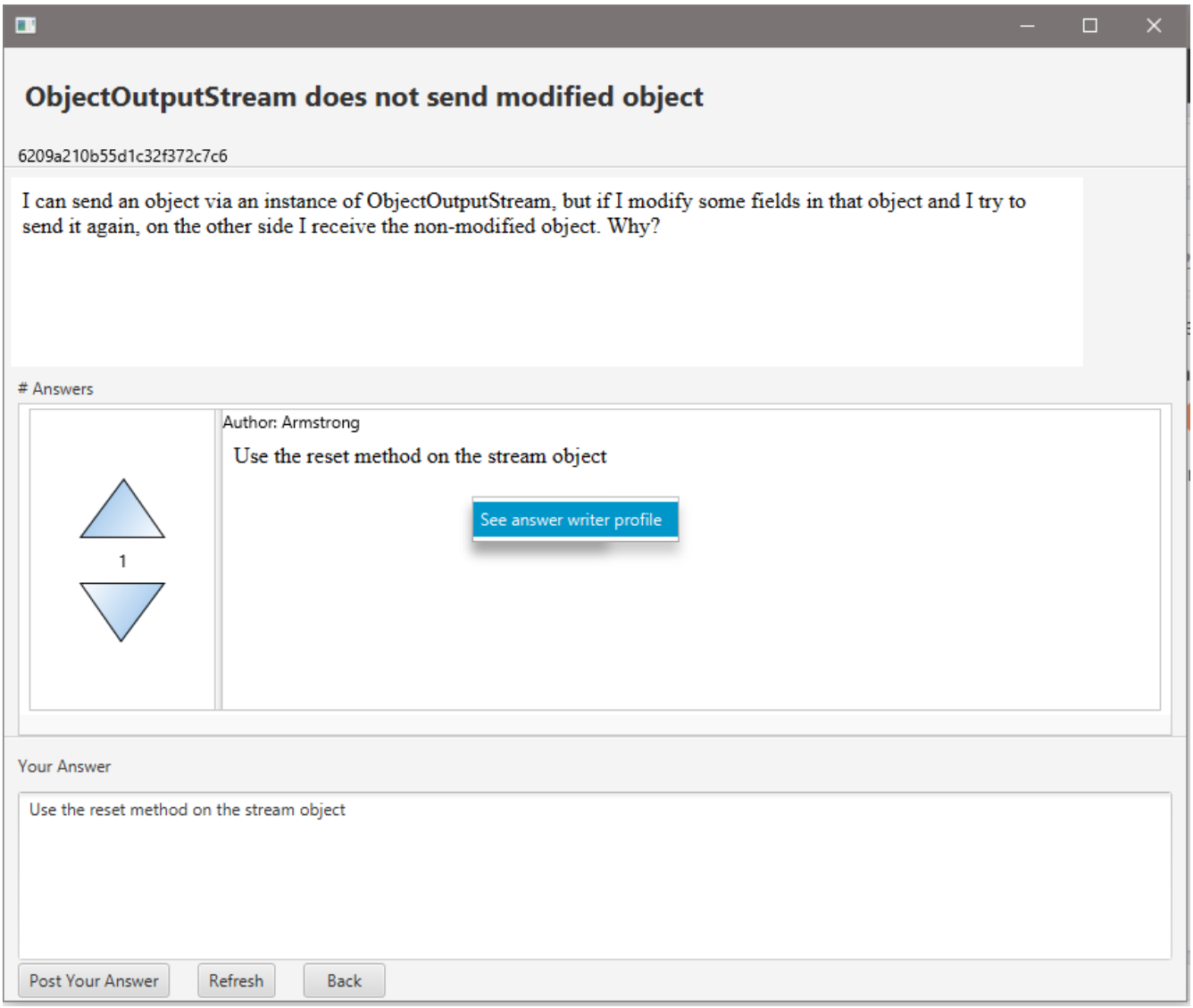


Figure 26: Check the answer writer profile

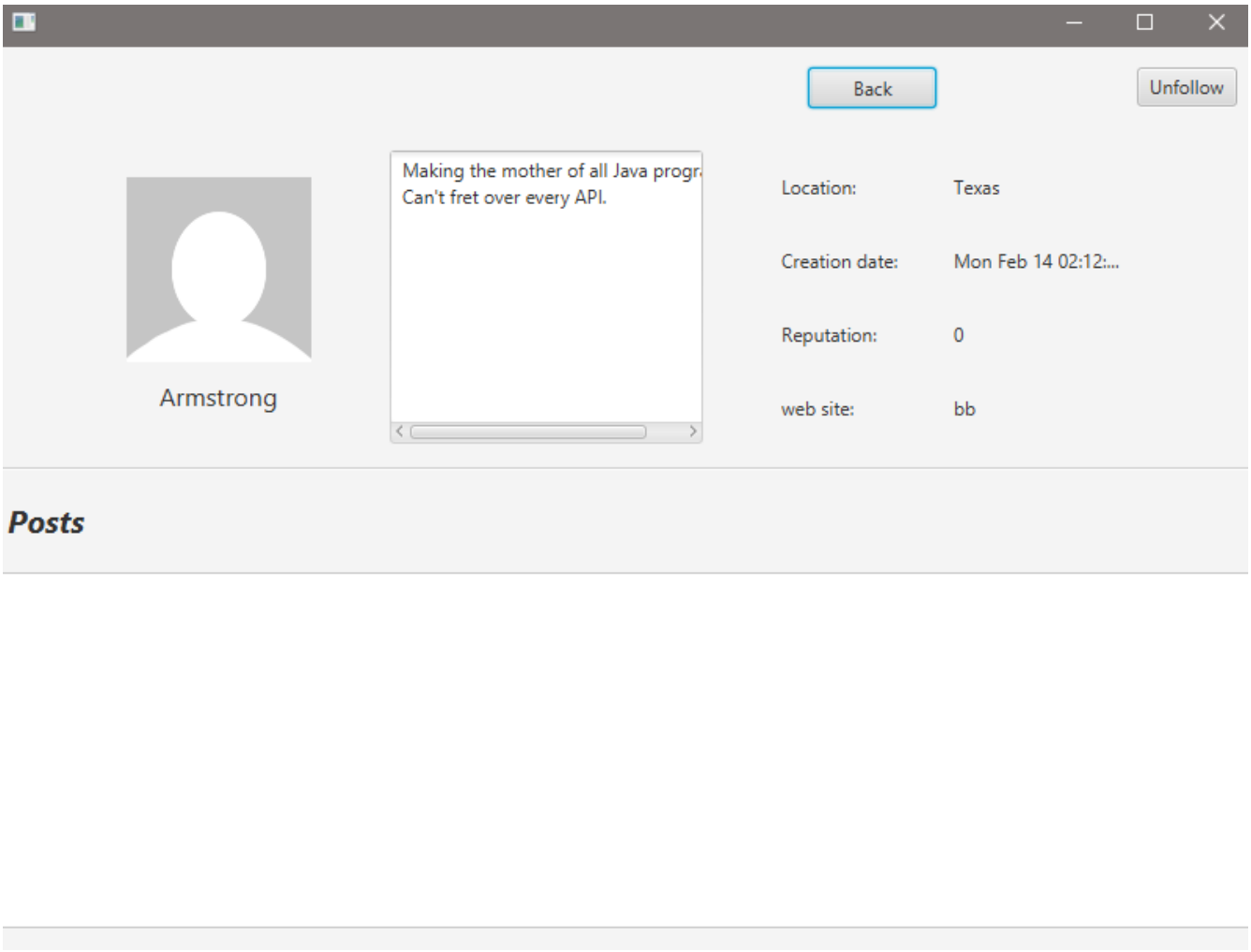


Figure 27: User who wrote the answer

4.11 Check user who wrote a post

1. Starting from the user profile (Fig. 10), go to the post search interface by clicking on the magnifying glass icon, and search any post
2. Right-click on the post itself, and then click the "See post writer profile" option (Fig. 28)
3. Details about the user who wrote the post will now appear on the screen (Fig. 29).

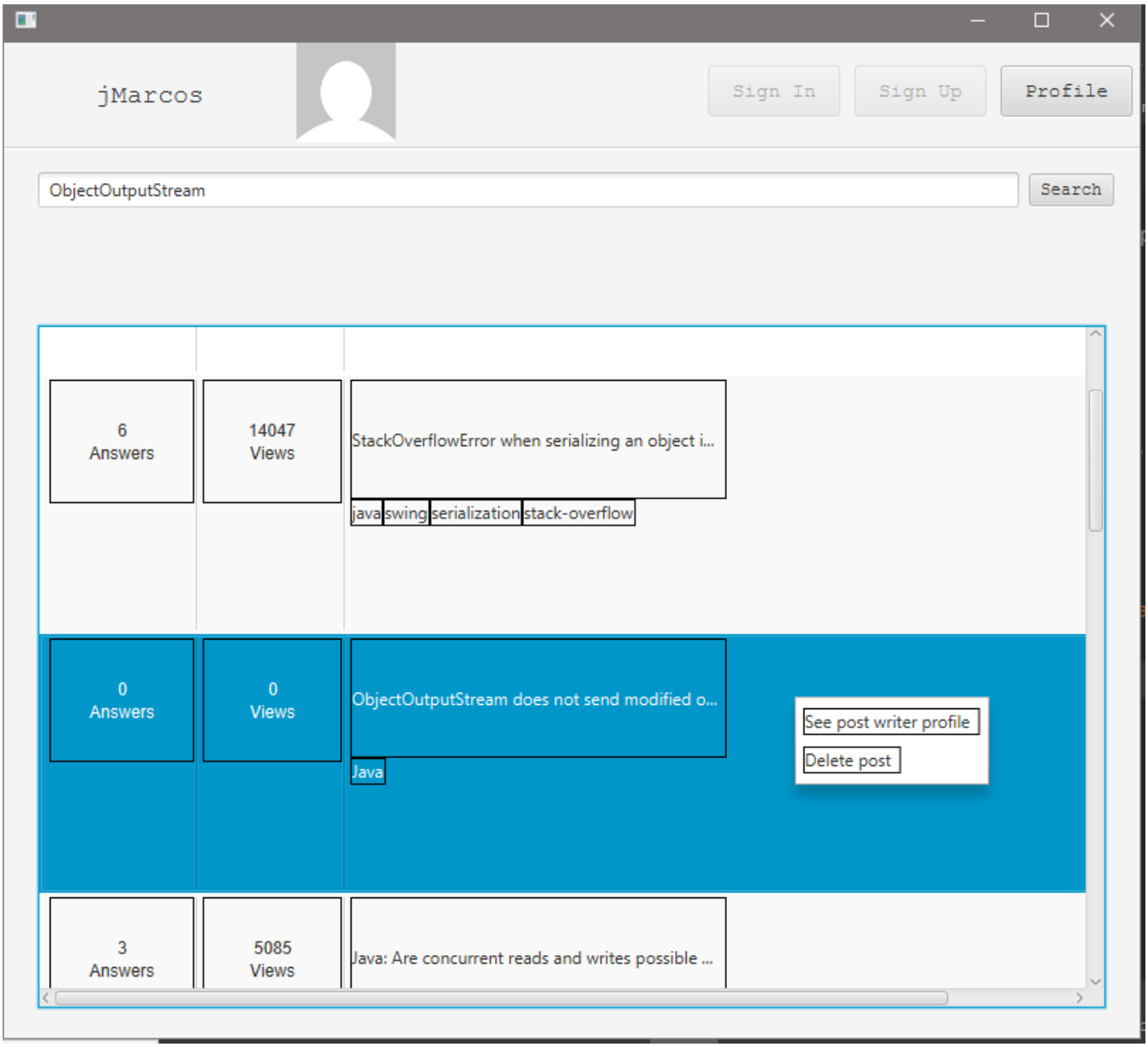


Figure 28: Check the post writer option

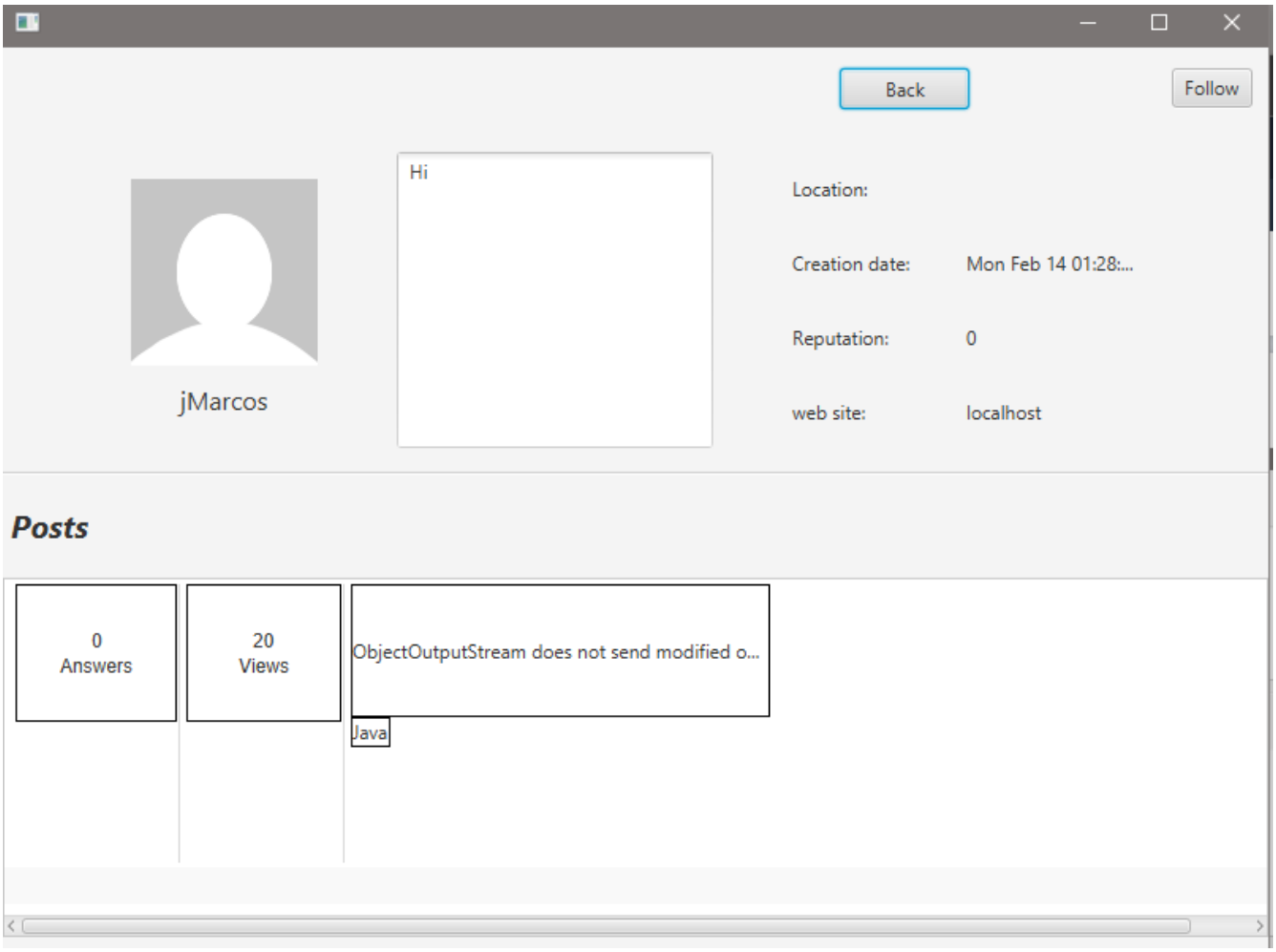


Figure 29: User who wrote the post

4.12 Follow an user

1. Starting from the user profile (Fig. 10), find any post or answer and go to their user profile (please refer to the "Check user who wrote a post/answer" section)
2. Click on the "Follow" button (Fig. 30)
3. Now you're following an user (Fig. 31)
4. In your own profile page, in the tab "Who I follow" it's possible to see all the user you follow (Fig. 32)
5. Similarly, in the tab "My Followers" you can see the users who follow you (Fig. 33).

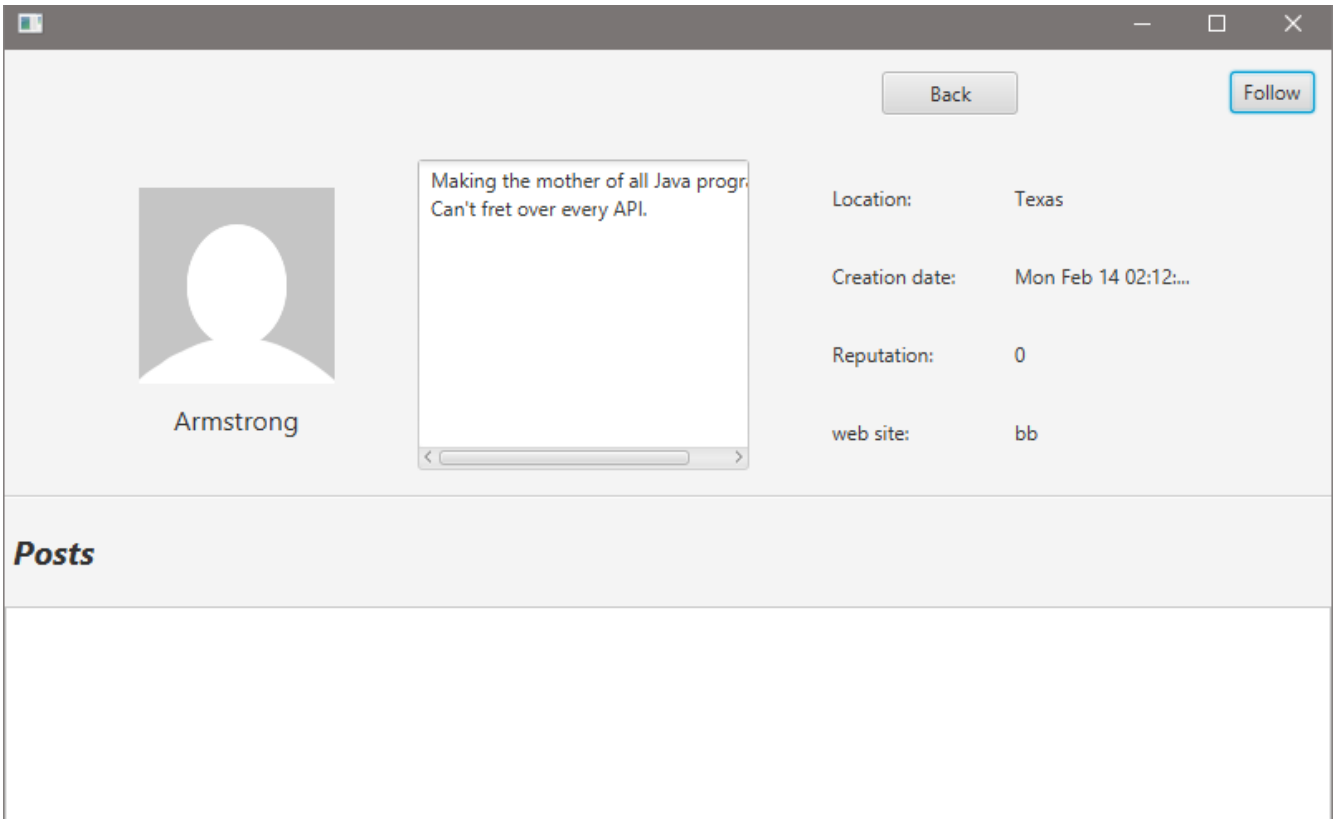


Figure 30: User profile to follow

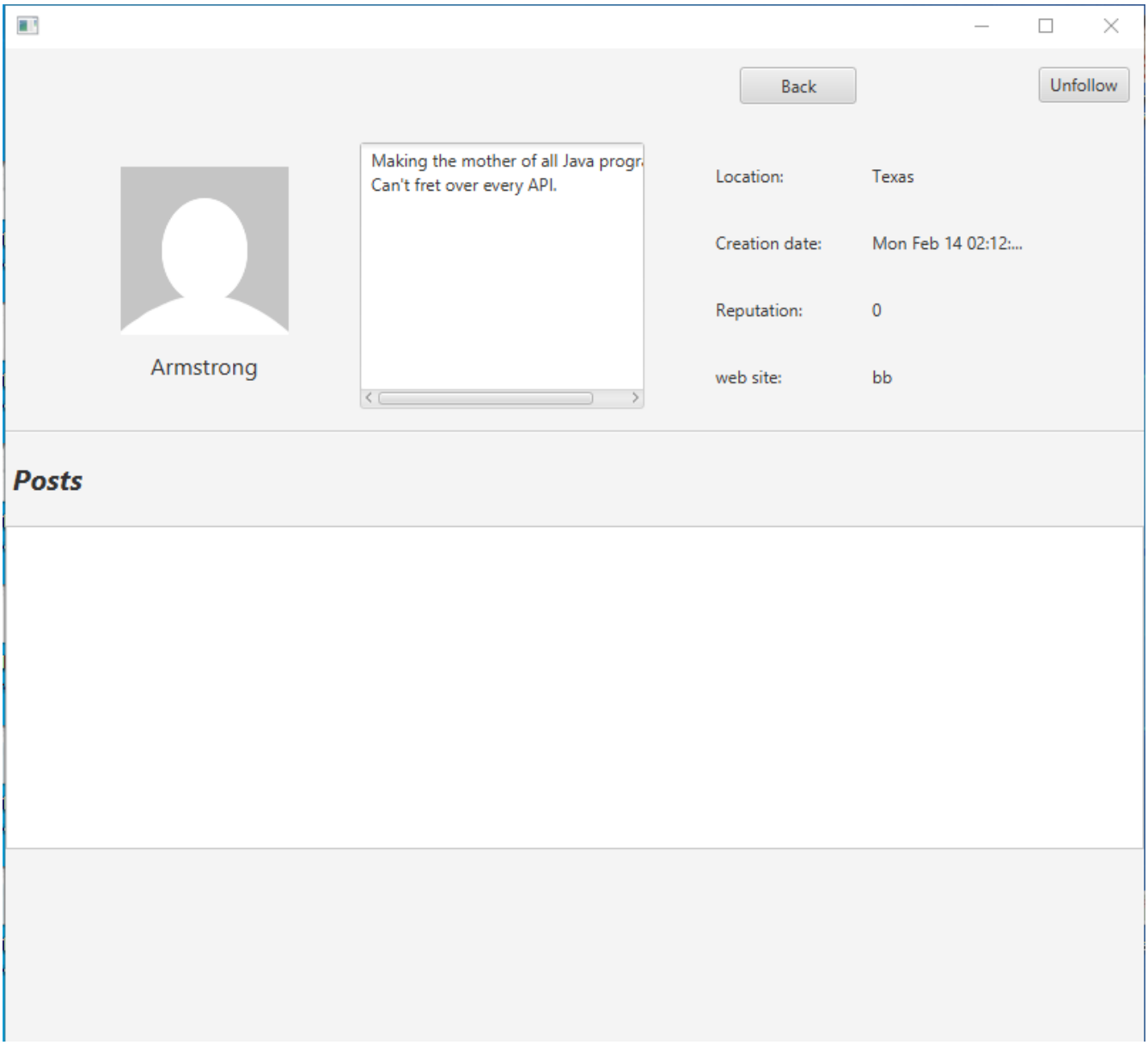


Figure 31: Following an user

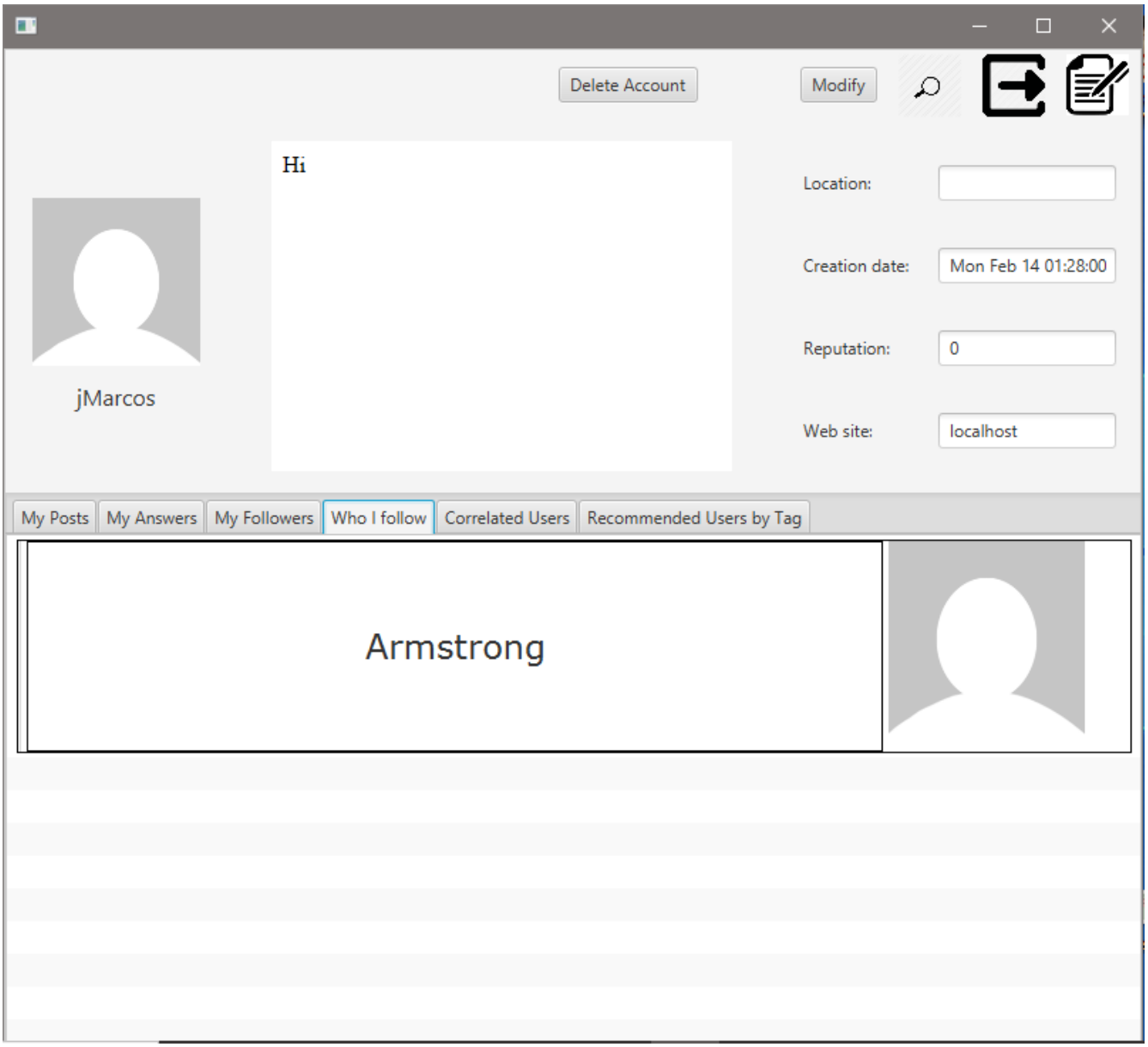


Figure 32: Checking your followed users

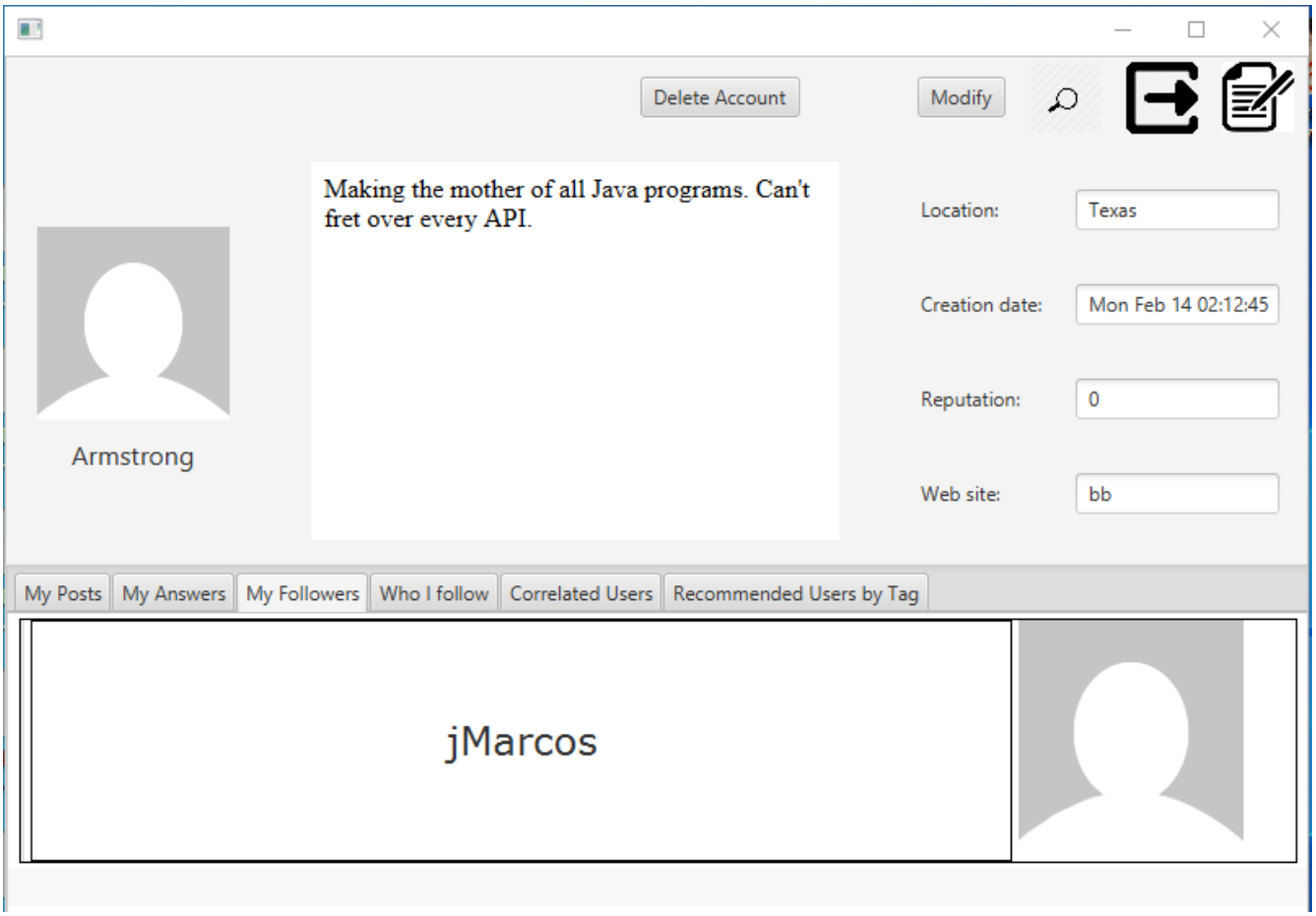


Figure 33: Checking your followers

4.13 Check your posts

1. Starting from the user profile (Fig. 10), click on the "My Posts" tab, and click on any post you want to check (Fig. 34)

2. You can now see the post details (Fig. 35).

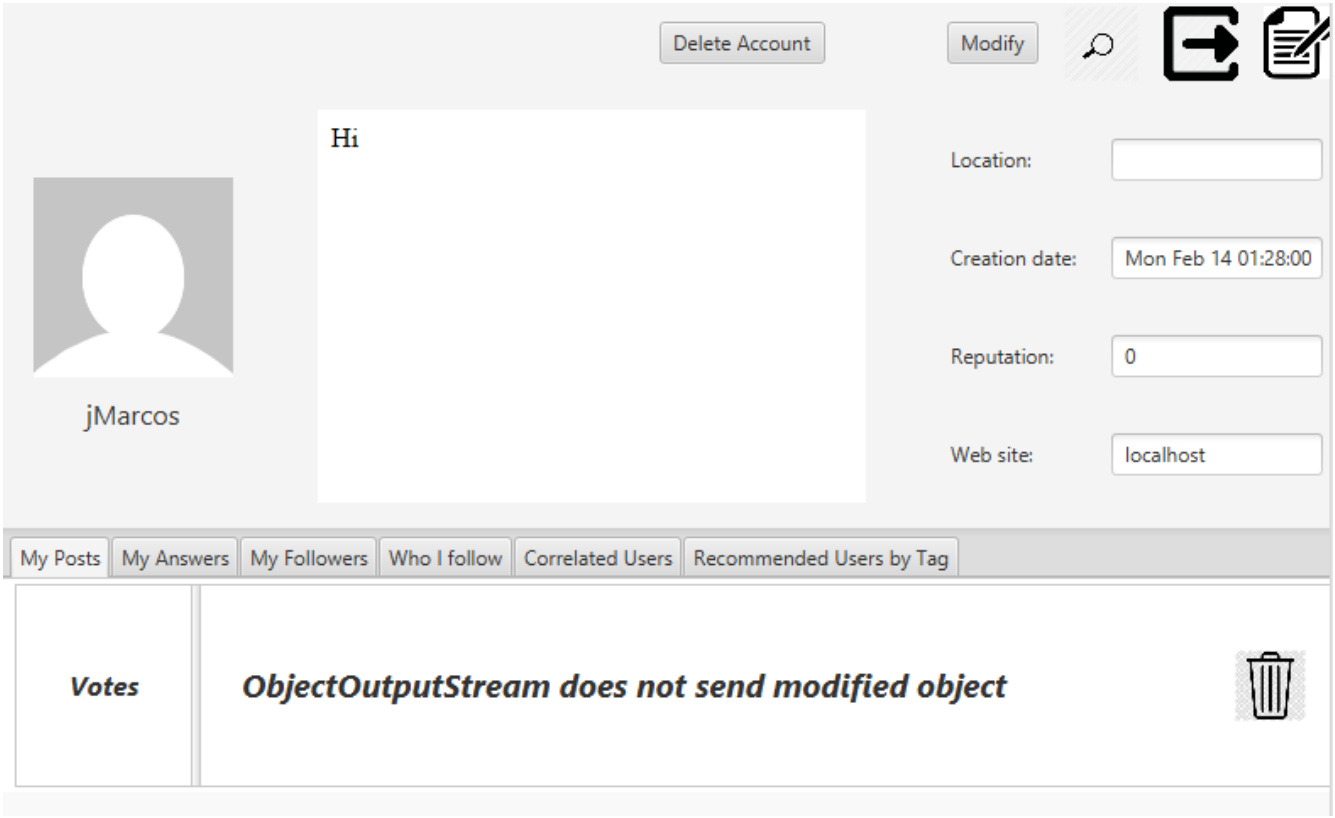


Figure 34: Checking your posts

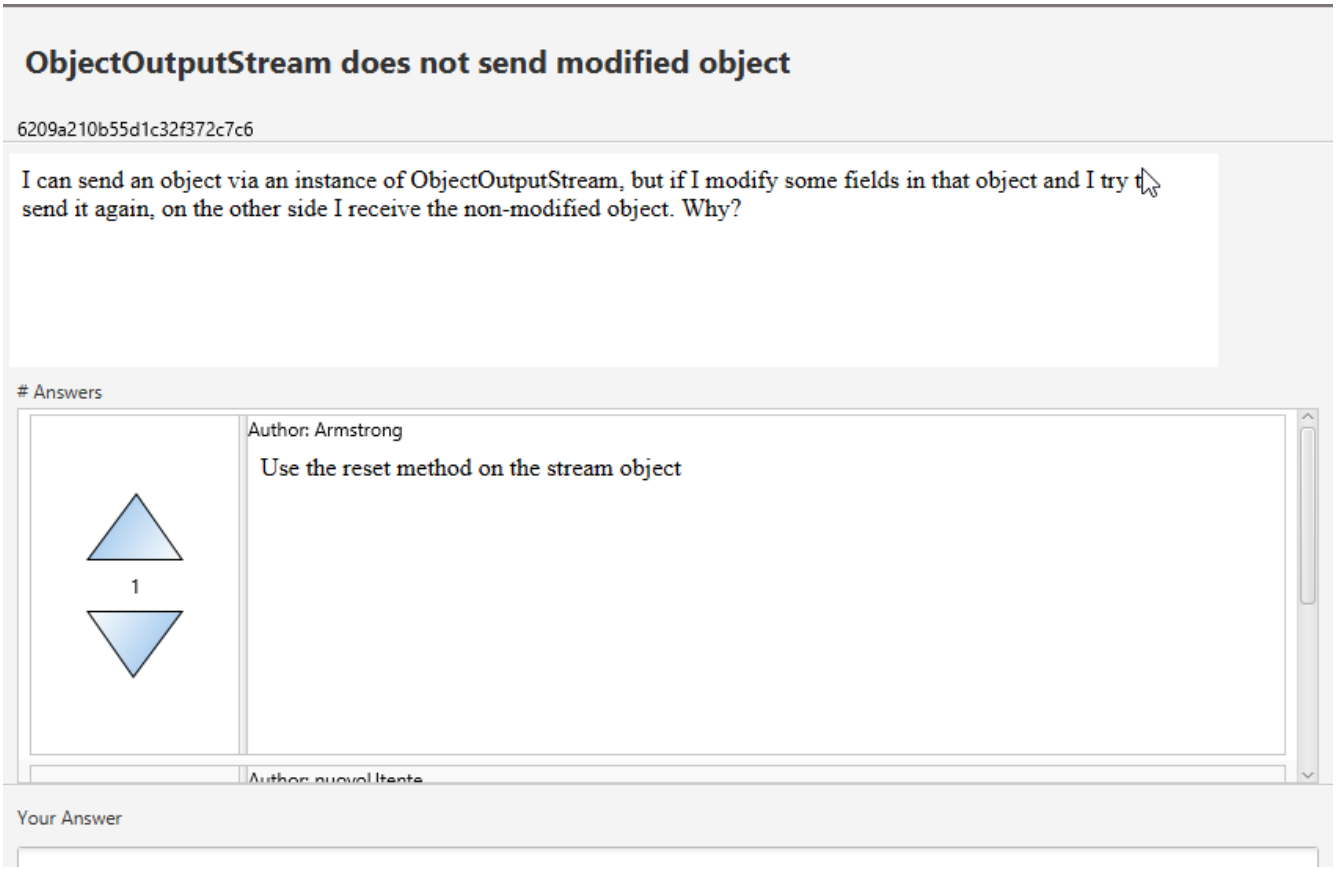


Figure 35: Post details

4.14 Check your answers

1. Starting from the user profile (Fig. 10), click on the "My Answers" tab, and click on any answer you want to check (Fig. 36)

2. You can now see the answer in the post (Fig. 37).

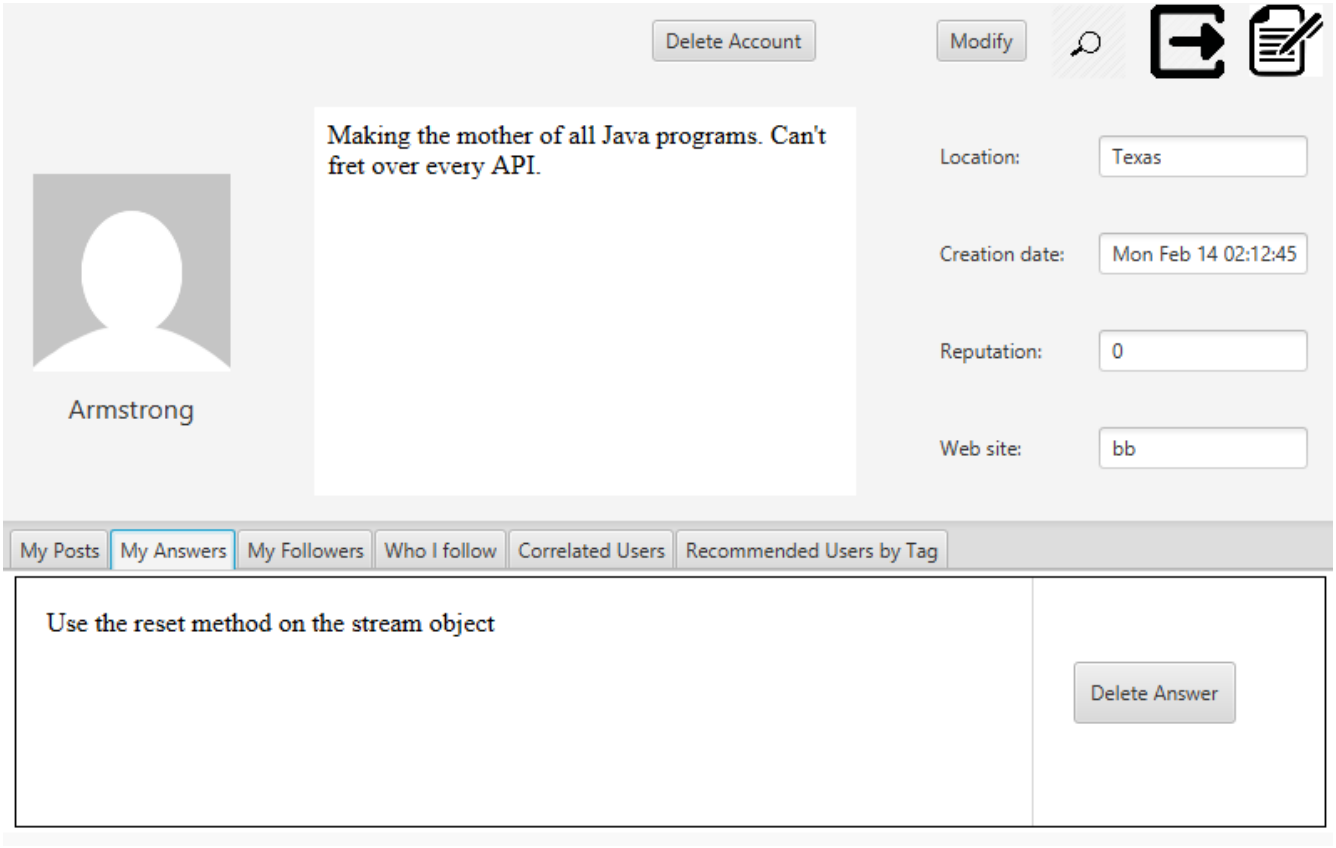


Figure 36: Checking your answers

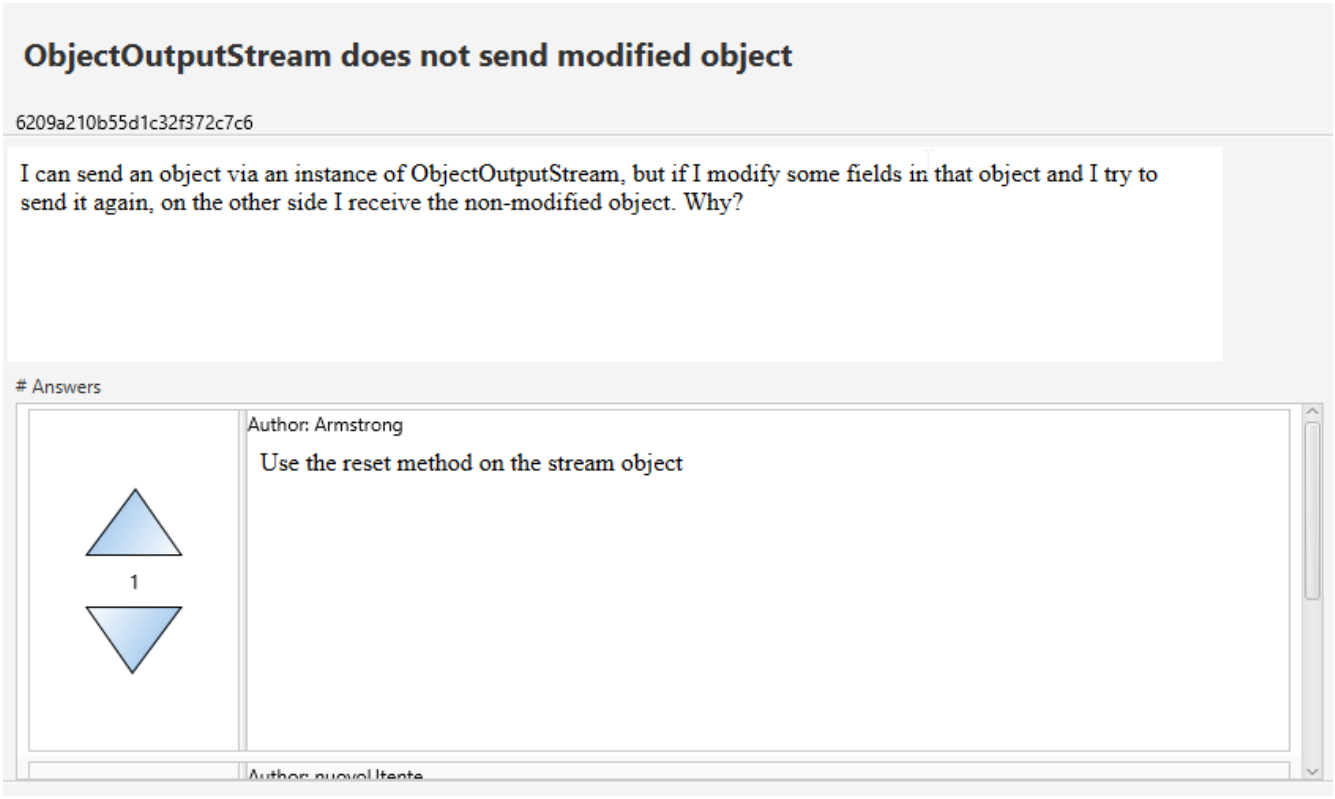


Figure 37: Answer details

4.15 Find a correlated user

For this example, let's suppose that the user "jMarcos" follows "Armstrong", and in a post a new user called "nuovoUtente" wrote a new answer.

1. Start by login with the "Armstrong" user
2. Find any post using the post search interface and left-click on it (Fig. 38)
3. You can now see the answer in the post (Fig. 39). Right-click it and go the profile of the answer writer (which is a different user from the mentioned ones
4. Follow the new user (Fig. 40)
5. After this, logout from "Armstrong" and login with "jMarcos". In the "Correlated Users" tab, you can see the new user "nuovoUtente" (Fig. 41).

Figure 38: Post search interface

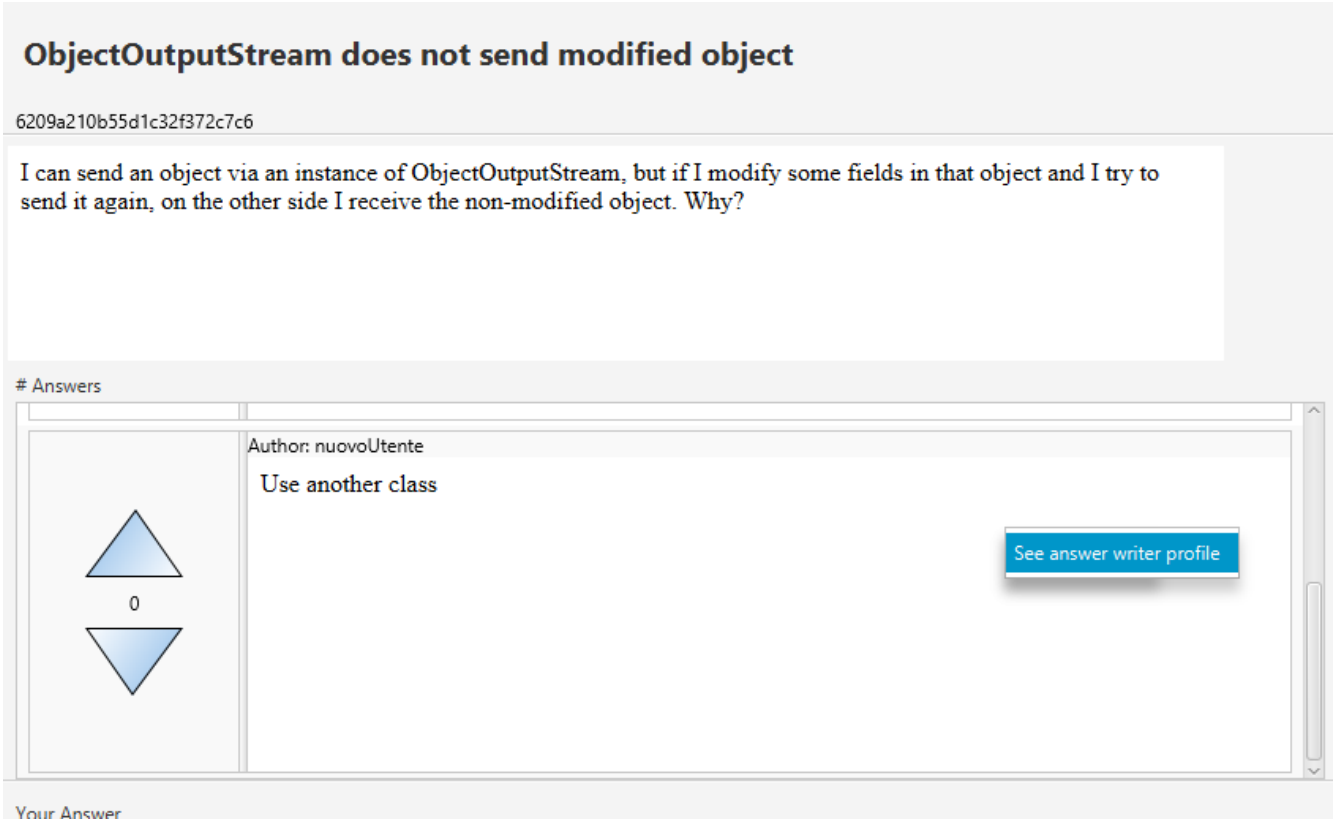


Figure 39: Check answer writer profile

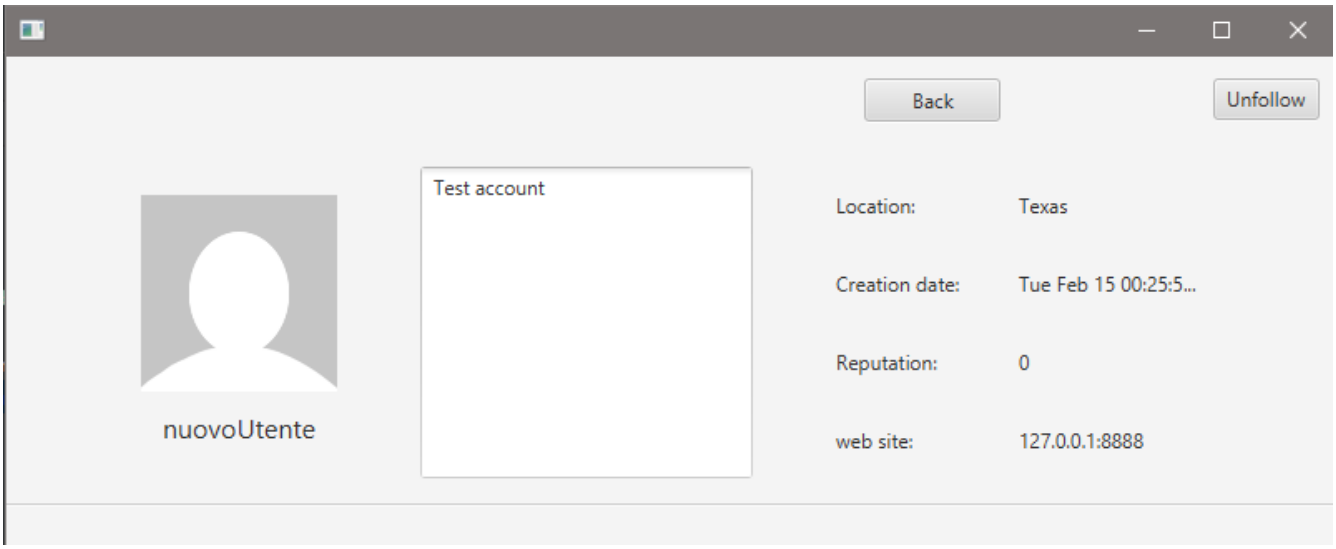


Figure 40: New user profile

Figure 41: Check correlated users

4.16 Remove a post

1. Starting from the user profile (Fig. 10), there are two alternatives:
    - Go to the post search interface, find the post you're looking for, then right-click it. IF you're the owner of the post or an admin, you can click on the "Delete post" option (Fig. 42)
    - IF you're the owner of the post, select the "My Posts" tab, and then click on the trash bin icon (Fig. 43)
2. Now the post has been deleted (Fig. 44).

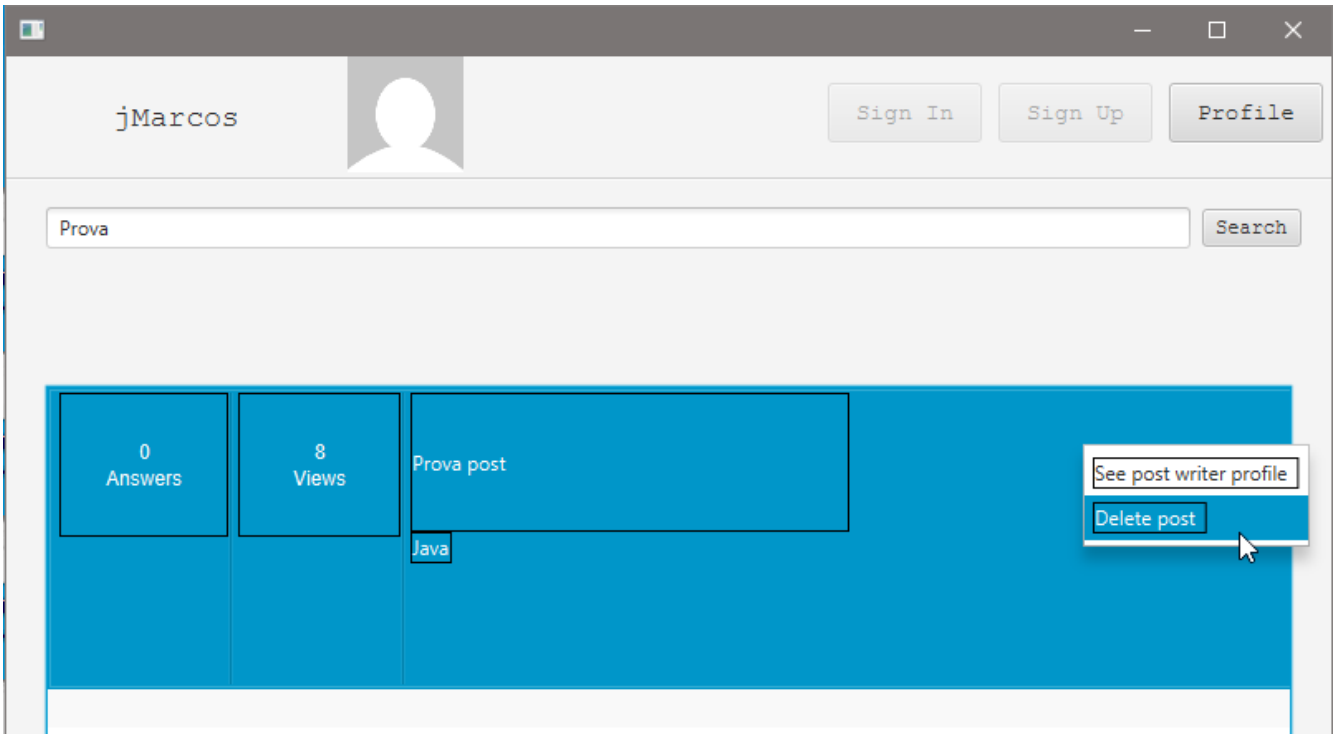


Figure 42: Delete post in the post search interface

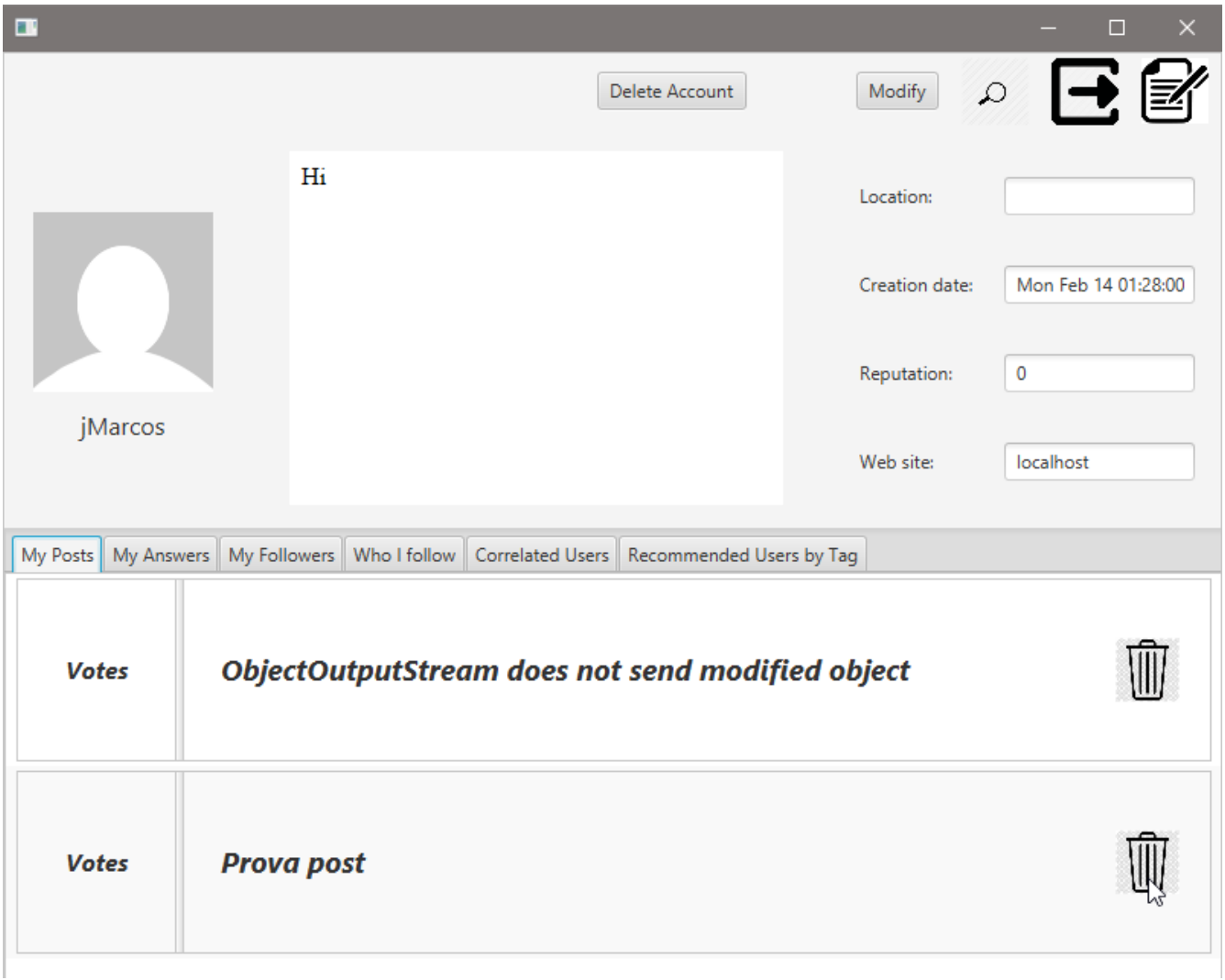


Figure 43: Delete post in the tab "My Posts"

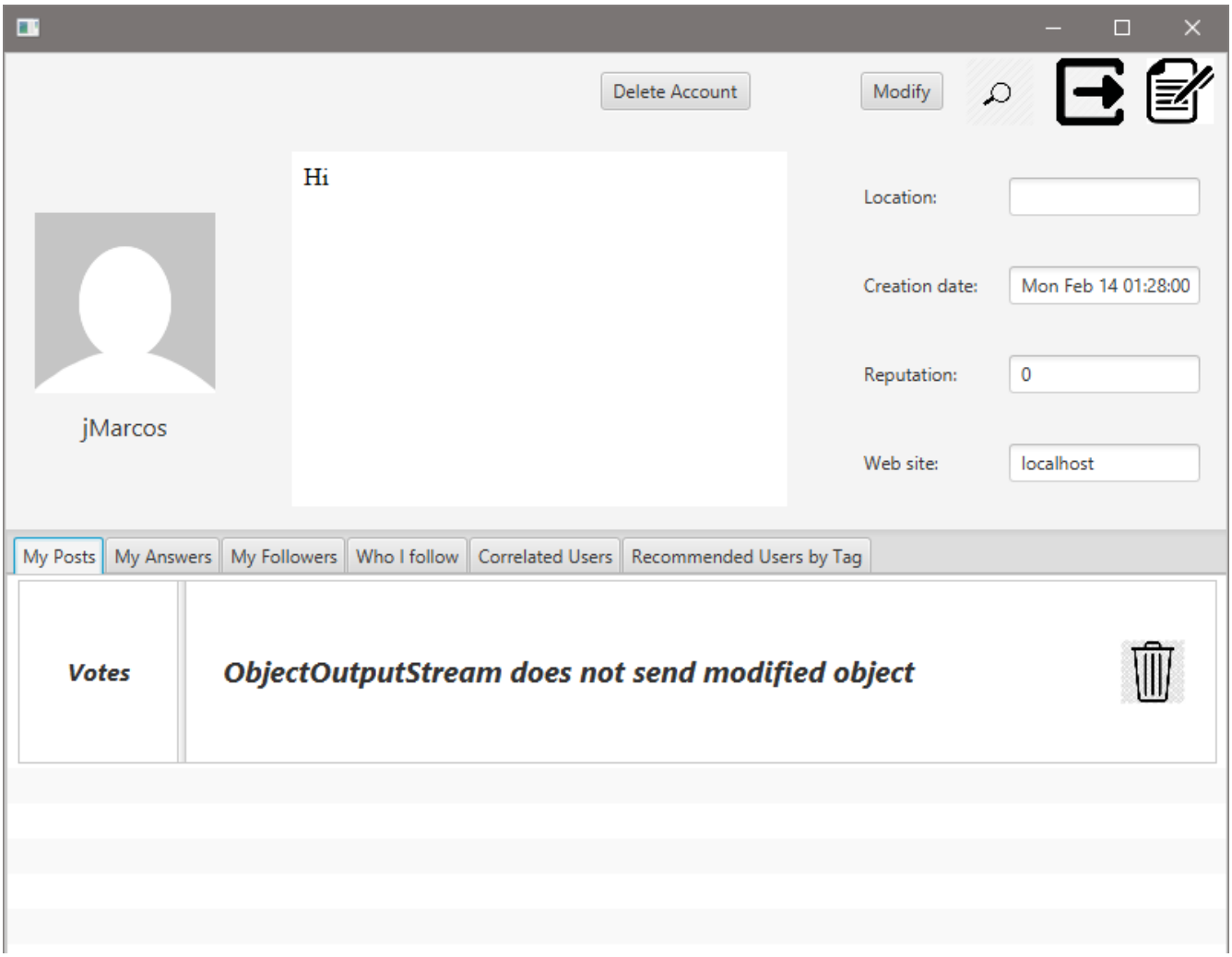


Figure 44: Post removed

4.17 Remove an answer

1. Starting from the user profile (Fig. 10), there are two alternatives:
   - Go to the post search interface, find the post you're looking for, then your answer, and finally right-click it. IF you're the owner of the answer or an admin, you can click on the "Delete answer" option (Fig. 45)
   - IF you're the owner of the answer, select the "My Answers" tab, and then click on the "Delete Answer" button (Fig. 46)
2. Now the answer has been deleted.

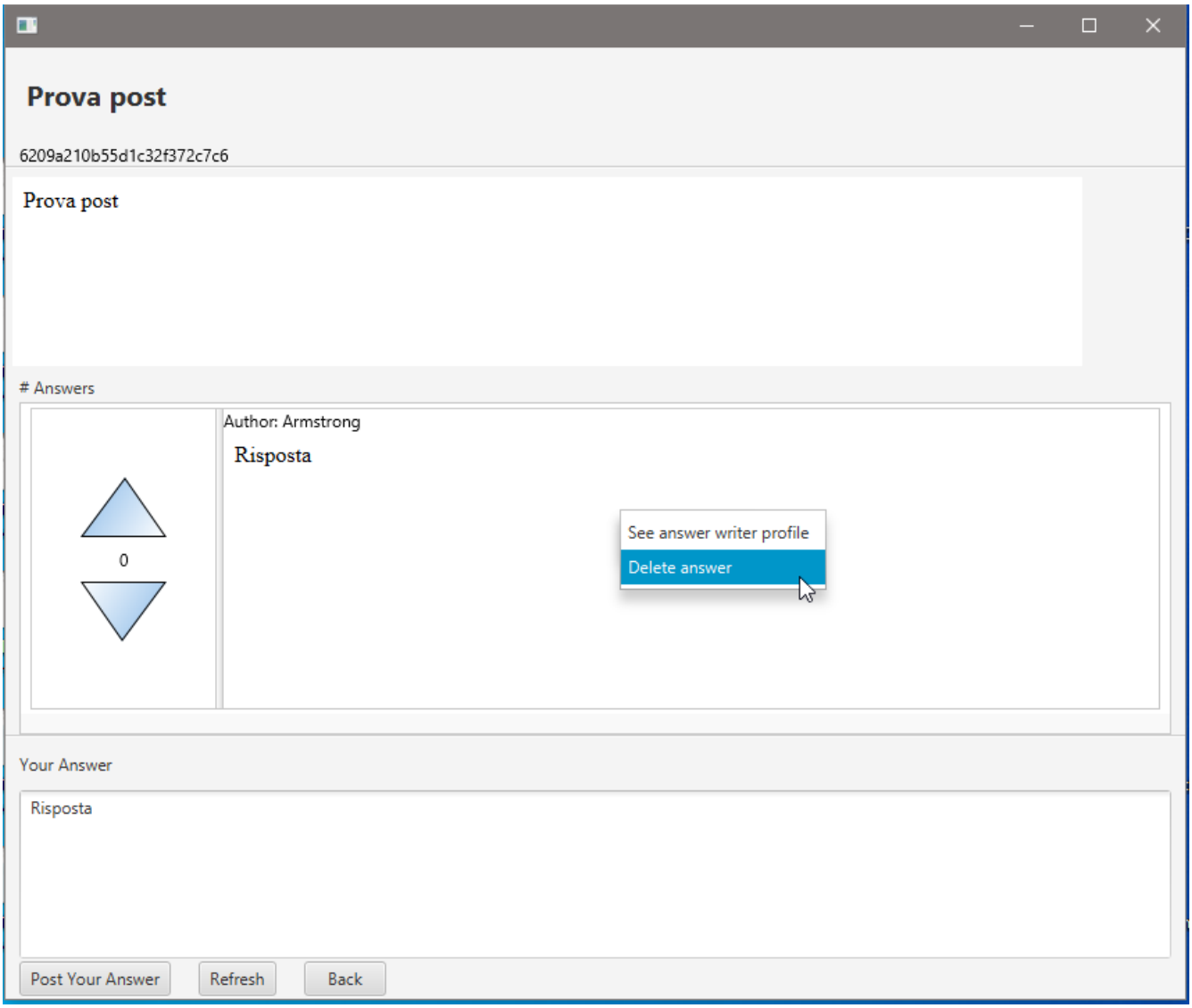


Figure 45: Delete answer in the post page

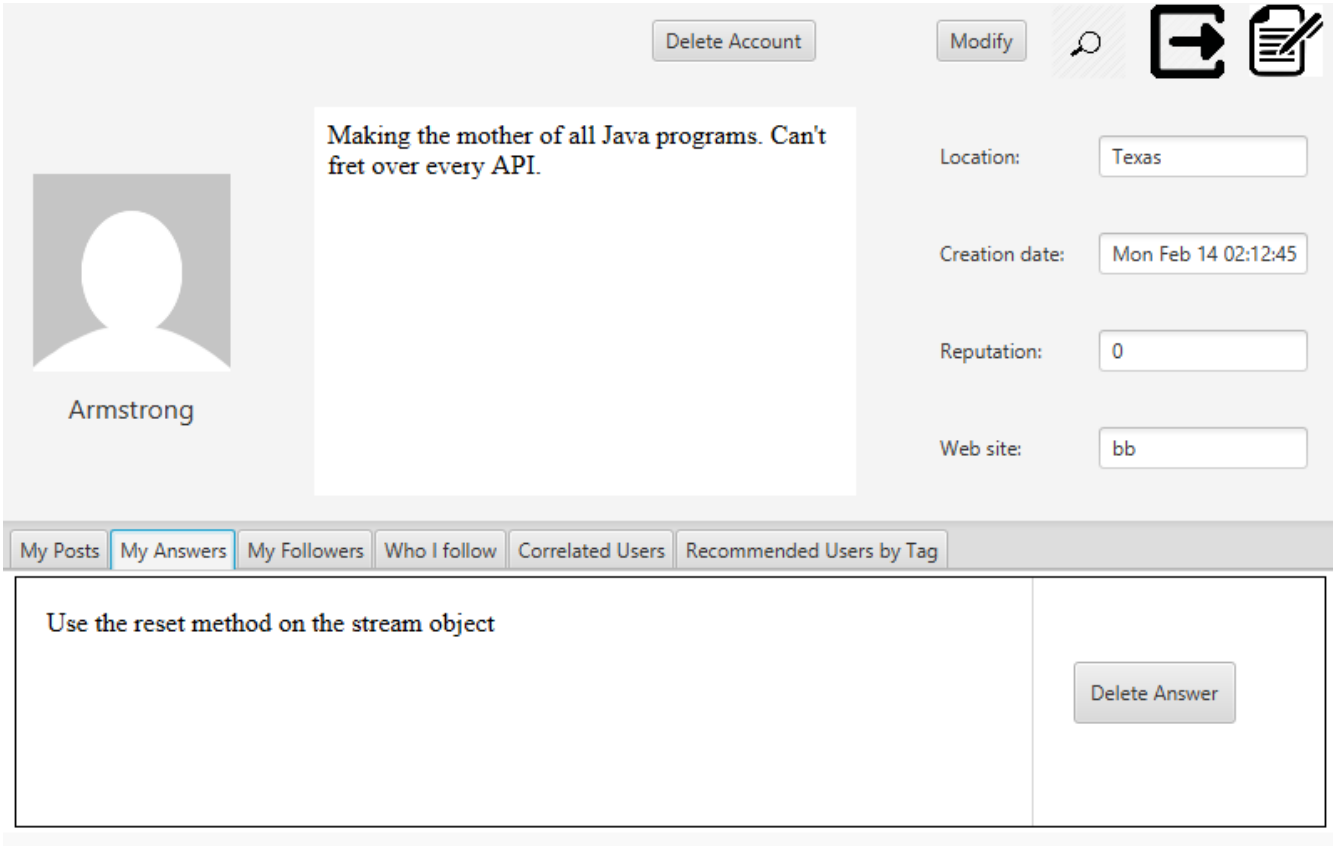


Figure 46: Delete answer in the "My Answers" tab

4.18 Delete your own account

1. Starting from the user profile, click on the "Delete Account" button (Fig. 47)
2. You'll be redirected to the home screen (Fig. 47).

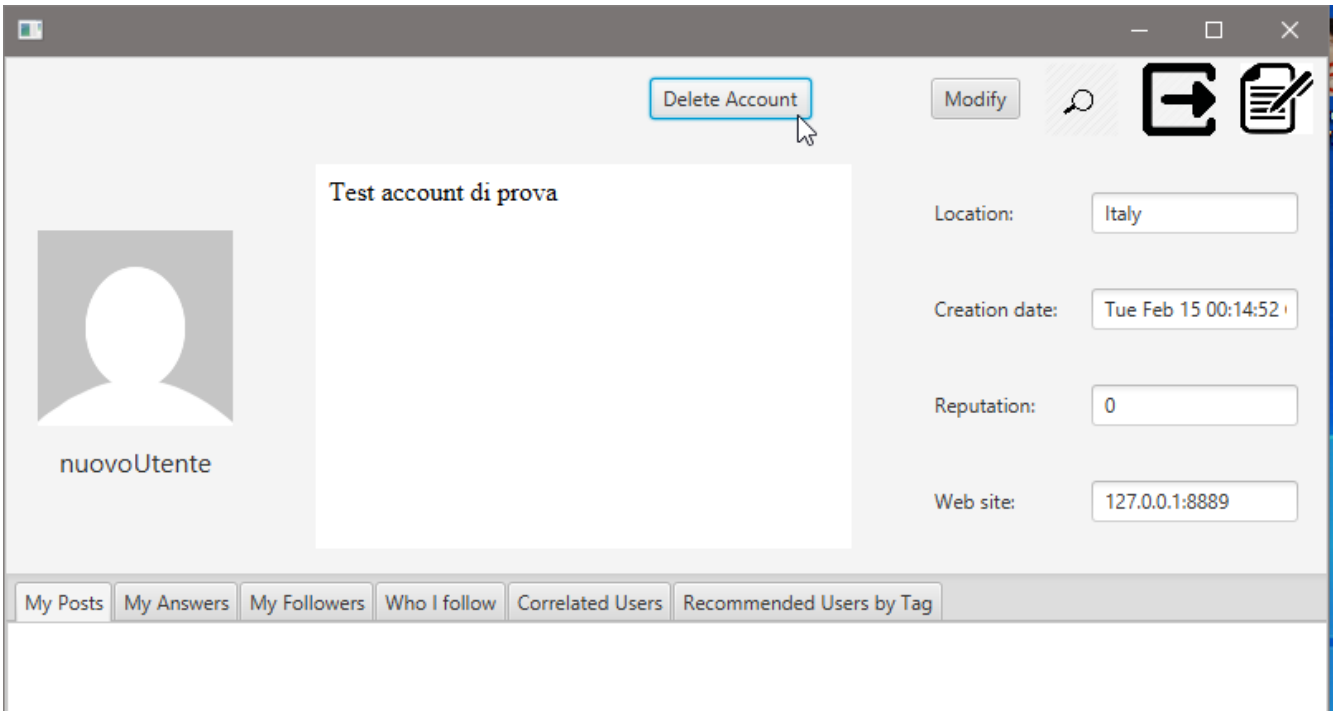


Figure 47: Delete account button

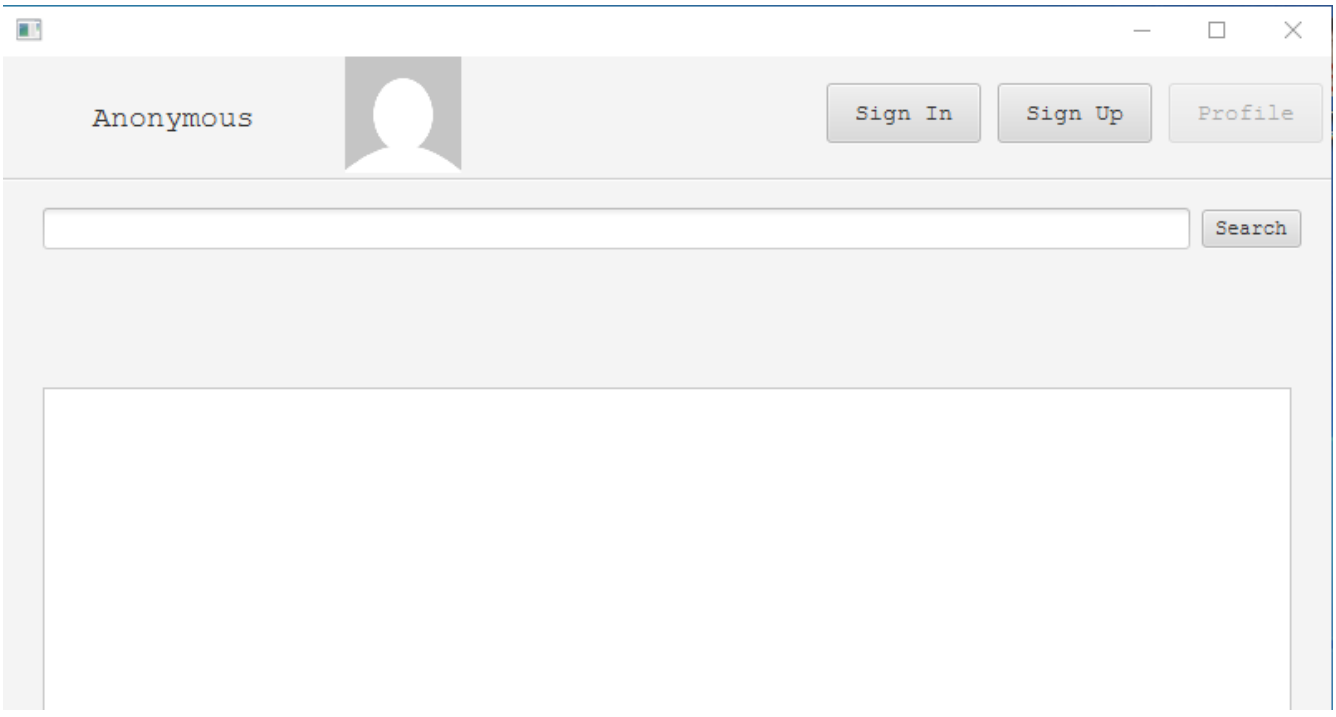


Figure 48: Back to the home screen after deletion

4.19 Delete another account

This operation can be done only as an admin.

1. Starting from the admin profile (Fig. 52), look for the user (find a post/answer the user has written, Fig. 49) and go to their profile (Fig. 50)
2. Click on the "Delete Account" button (Fig. 51)
3. You'll be redirected to the admin profile (Fig. 52)

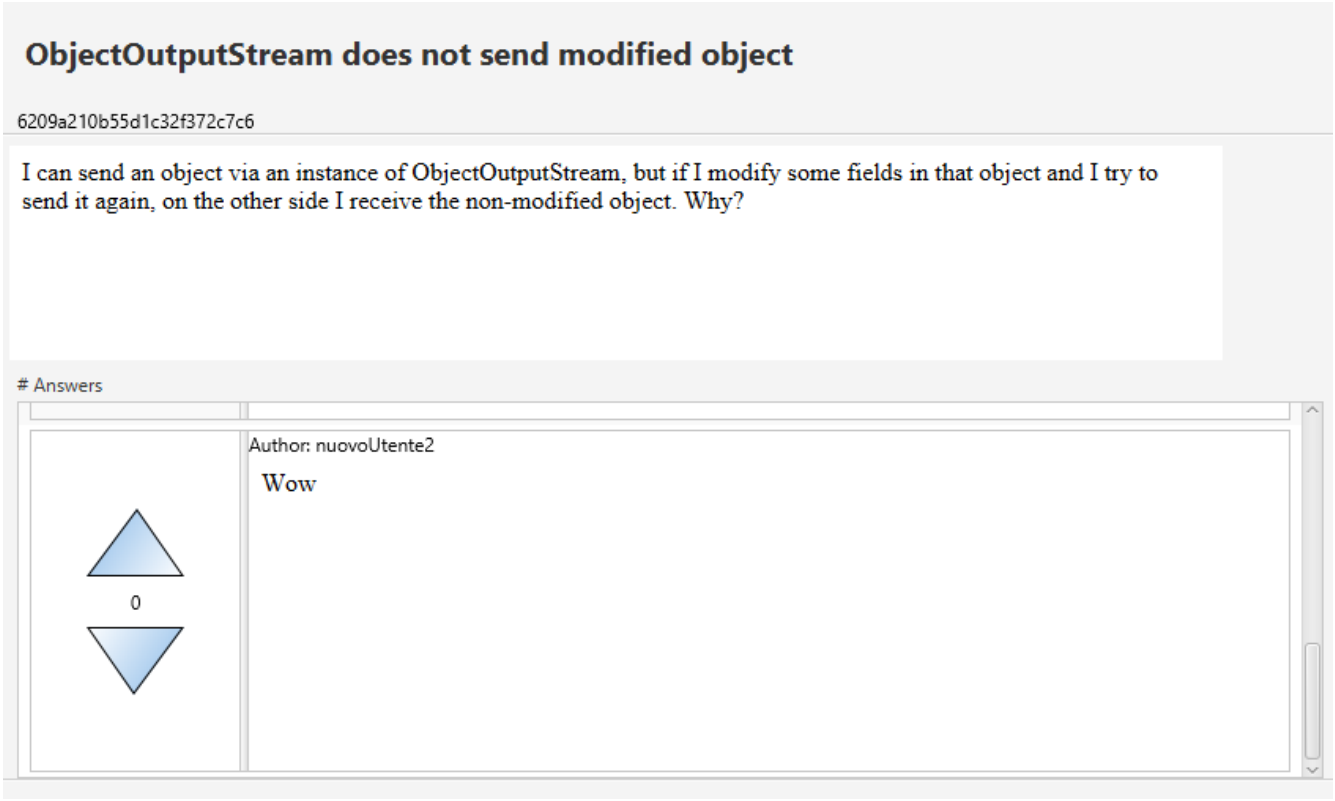


Figure 49: Post where the user is located

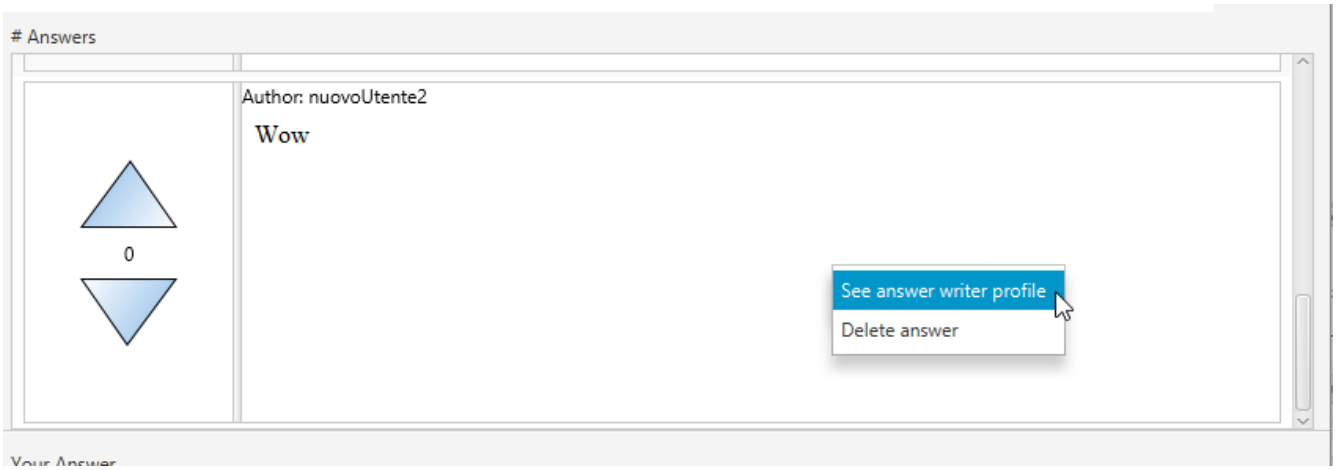


Figure 50: Going to their profile

Figure 51: Deleting the user account

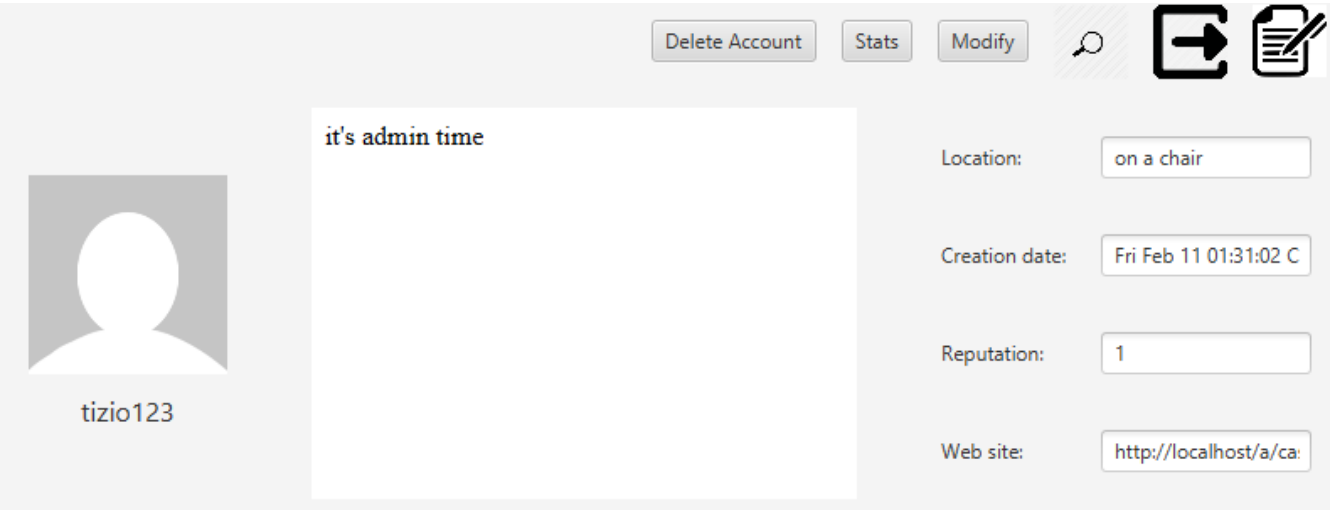


Figure 52: User removed

4.20 Get to the analytics section

This operation can be done only as an admin.

1. Starting from the admin profile (Fig. 53), click on the "Stats" button
2. In this tab different analytics results can be seen (starting from top-left to bottom): expert users w.r.t. a single tag (more on this later), the most popular users (the ones with the highest reputation), a list of hot topics for the top users (the most followed users), the most answered posts of top users (as before), and a chart containing the most popular tags (Fig. 54)
3. In order to populate the "Experts by Tag" table, first write a tag inside the textbox on the bottom side of the table, and then click "Search" (Fig. 55)
4. You can even click on one user to see his/her profile (Fig. 56 and Fig. 57).

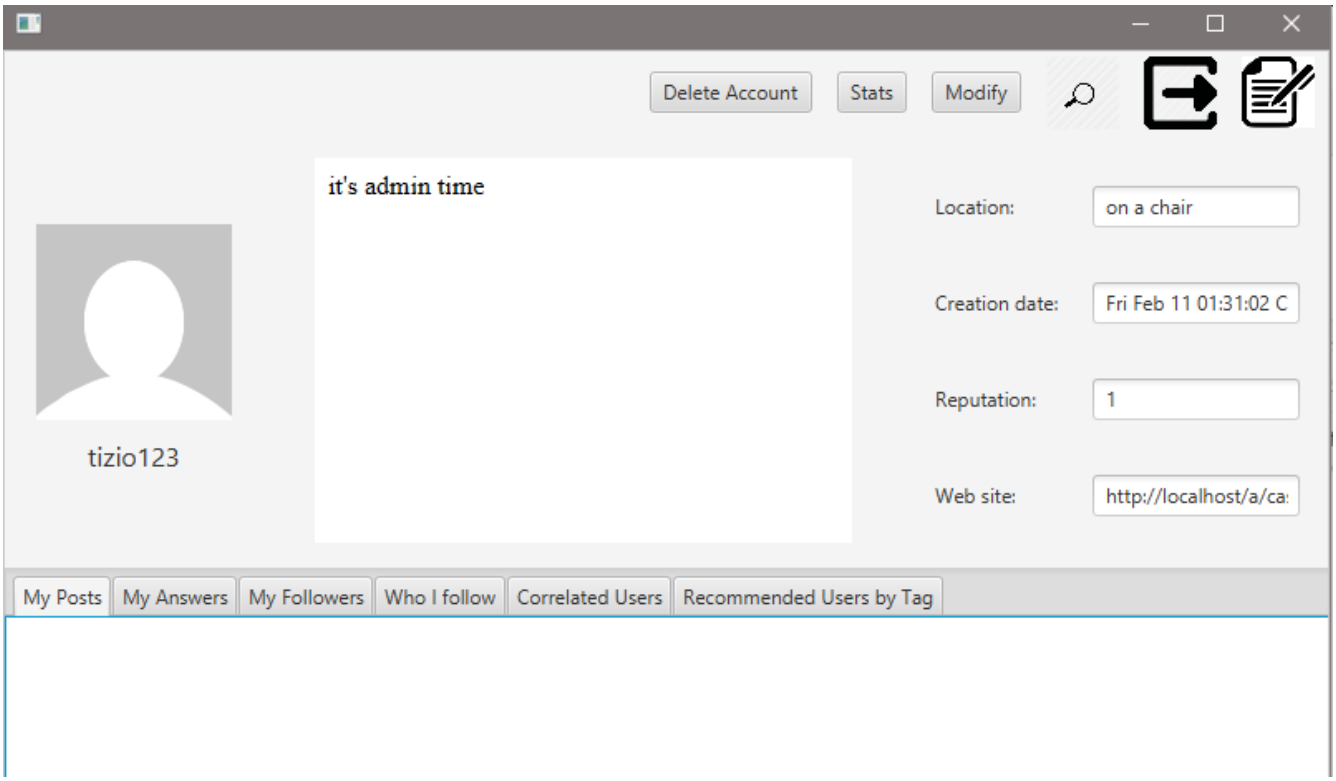


Figure 53: Admin profile

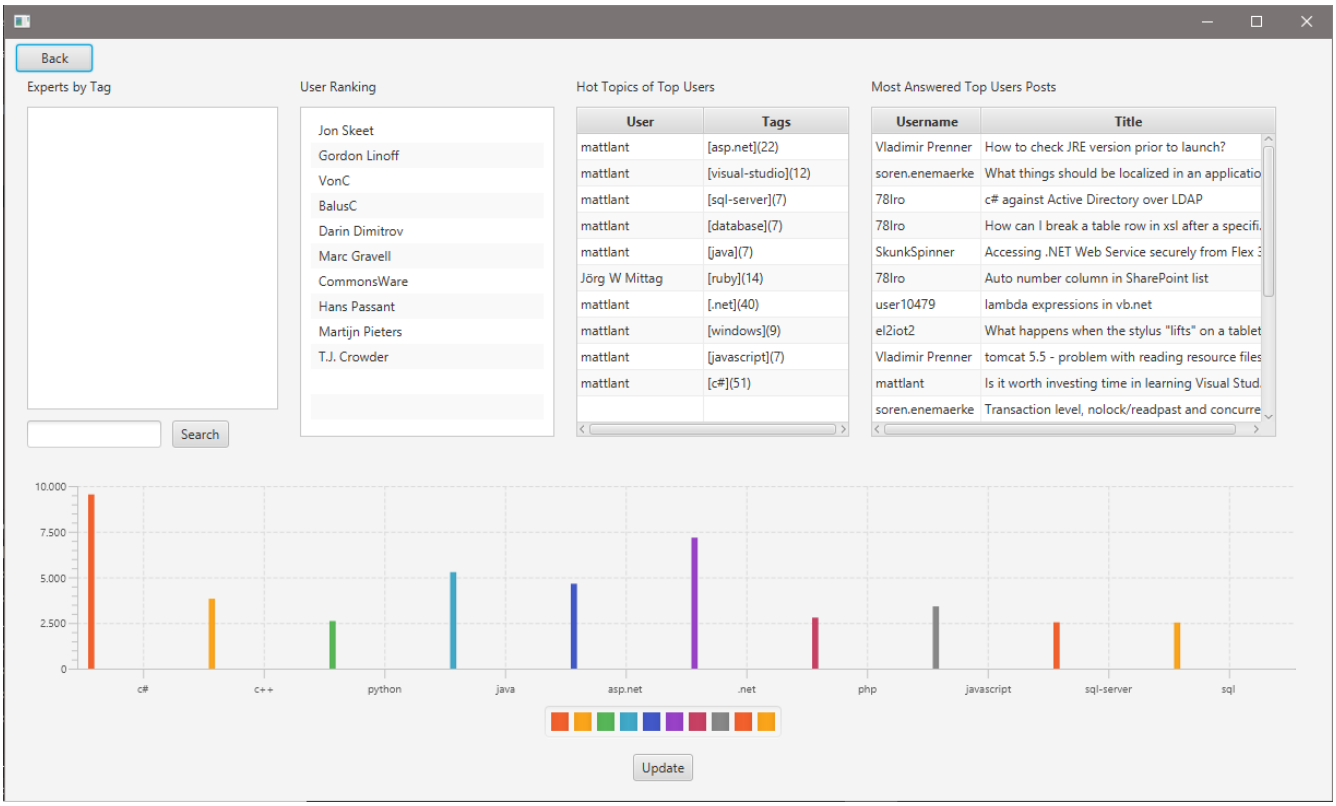


Figure 54: Analytics page

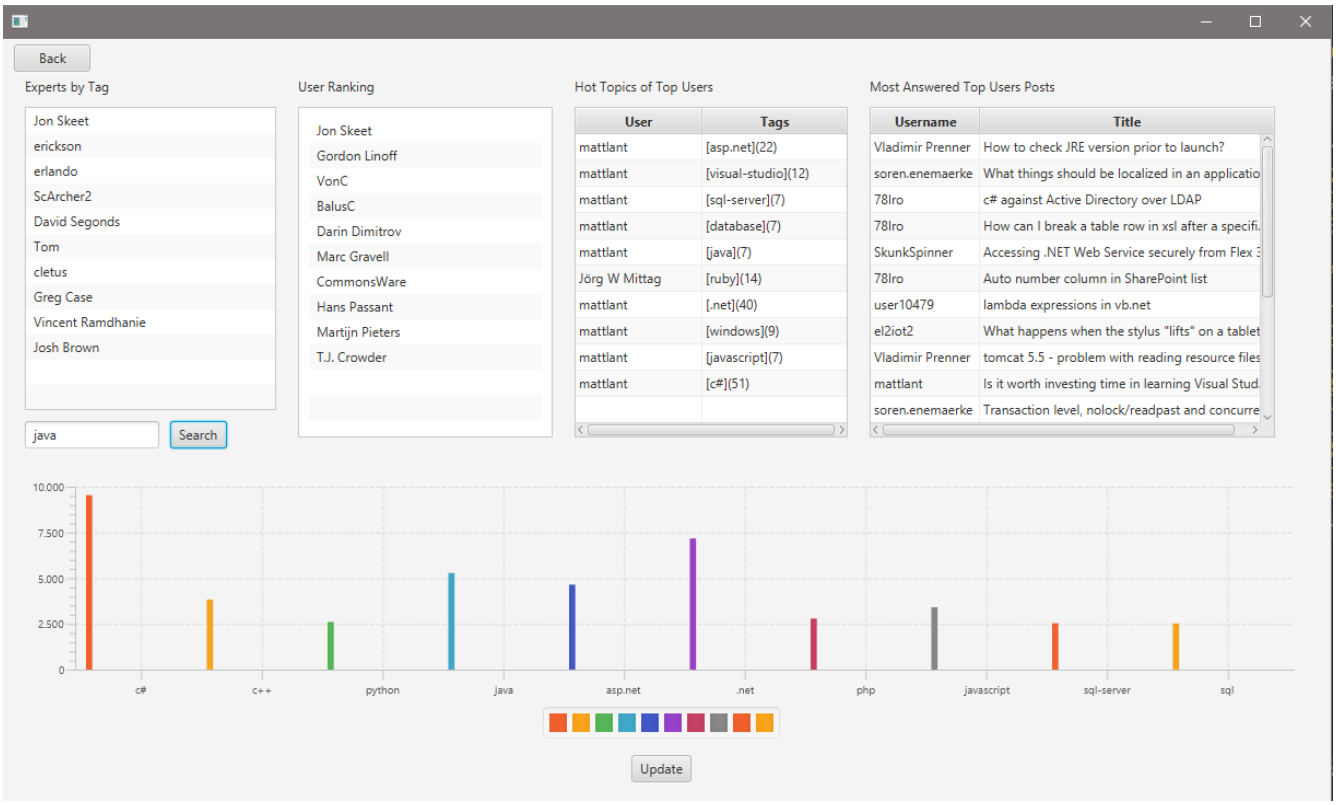


Figure 55: Experts by Tag table populated

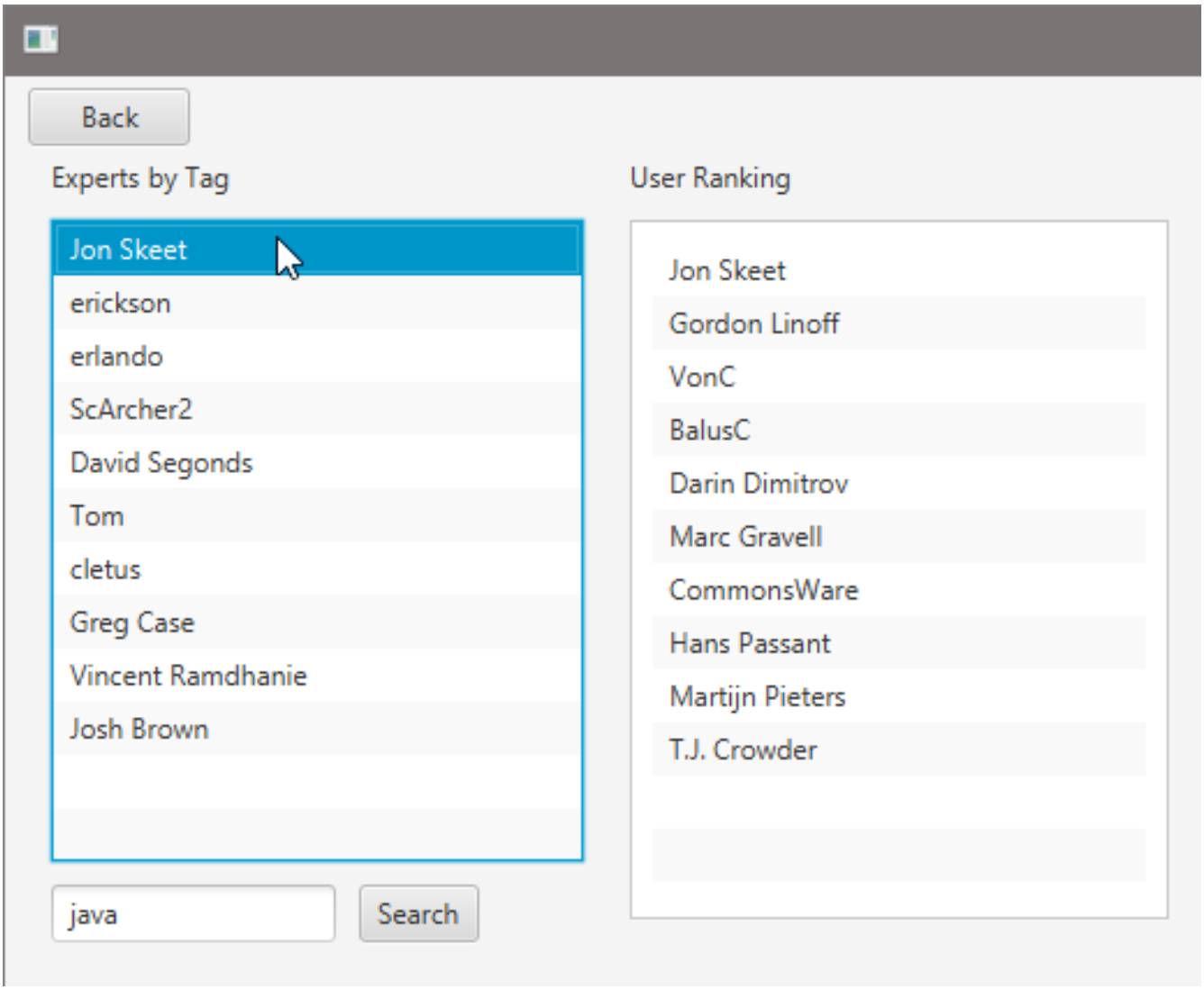


Figure 56: Clicking on one user

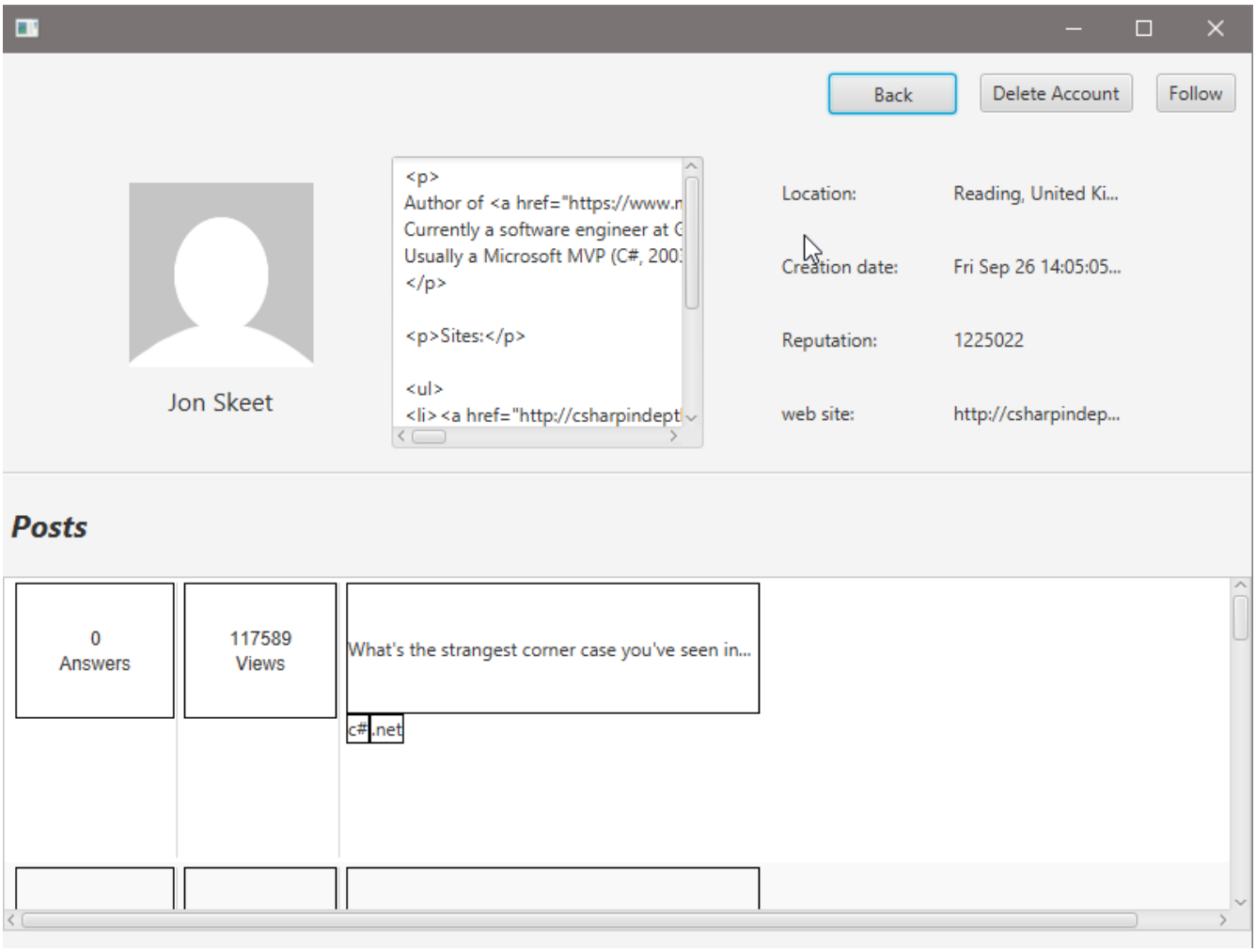


Figure 57: User profile