

Assignment #7—SteamTunnel

Due: 5PM (not 11AM) on Wednesday, June 6th

Note: No late days (free or otherwise) may be used on this assignment
This assignment may be done in pairs (which is optional, not required)

Based on an assignment by Mehran Sahami.

Social network applications (such as Instagram, LinkedIn, and Snapchat) are used by billions of people. They are immensely popular particularly because of the power of the internet to connect people. Your job for this assignment is to create a program that keeps track of a simple, internet based, social network: SteamTunnel.¹

Background

Internet-connected applications like Steam Tunnel are actually two separate programs: the “server” (aka the cloud, aka the backend) and the “client” (aka the frontend). The server stores all of the data for your social network. The client runs on a user’s device. (Fun fact: did you know Facebook’s datacenters and Android phones both run Java?).

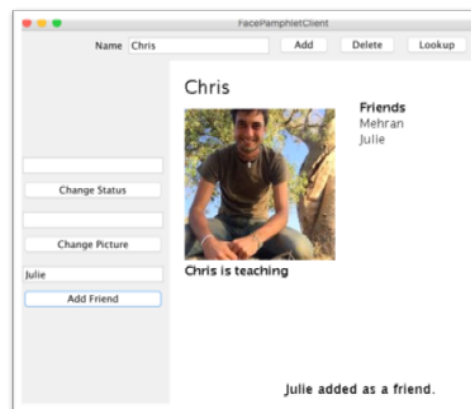
SteamTunnel Server

```
Starting server on port 8000...
addProfile (name=Mehran)
=> success
addProfile (name=Chris)
=> success
addProfile (name=Chris)
=> Error: Database already contains Chris.
setStatus (name=Chris)
=> none
setStatus (name=Chris, status=teaching)
=> success
setStatus (name=Chris)
=> teaching
addFriend (name2=Mehran, name1=Chris)
=> success
getFriends (name=Chris)
=> [Mehran]
addProfile (name=Julie)
=> success
setImg (name=Julie)
=> none
setStatus (name=Julie)
=> none
getFriends (name=Julie)
=> []
setImg (img=JulieZ.jpg, name=Julie)
```

Communicate via
the internet



SteamTunnel Client



¹ Monica Lam, a professor in the Computer Science department researches *federated* web applications (among other things). A *federated* social network is one where many people / institutions make many social networks. If the many social networks agree on a common method of communication, users could connect regardless of which social network they joined, just like with email. A nice result of such a system is that one single company (eg Facebook) wouldn’t have total power over online social interactions. For simplicity making your network federated is an extension and not required.

The client and the server work as a team to provide a persistent internet experience. They are constantly sending messages back and forth to each other. The client sends requests to the server (such as `setStatus` or `getStatus`) and the server updates its data and returns responses back to the client. All of these communications are sent over the internet.

In this assignment, you are going to write **the server** of the Steam Tunnel application. As an extension you can optionally write a **client** program that communicates with your server. This assignment has you practice the essential concepts that we have learned in the second half of the course in addition to teaching you about internet applications.

The SteamTunnel server program keeps tracks of the information about the social network; specifically, all users' profiles. A profile includes: a name, an optional image for a profile picture, an optional "current status" (a `String` indicating what activity the owner of that profile is currently engaged in), and a list of friend names.

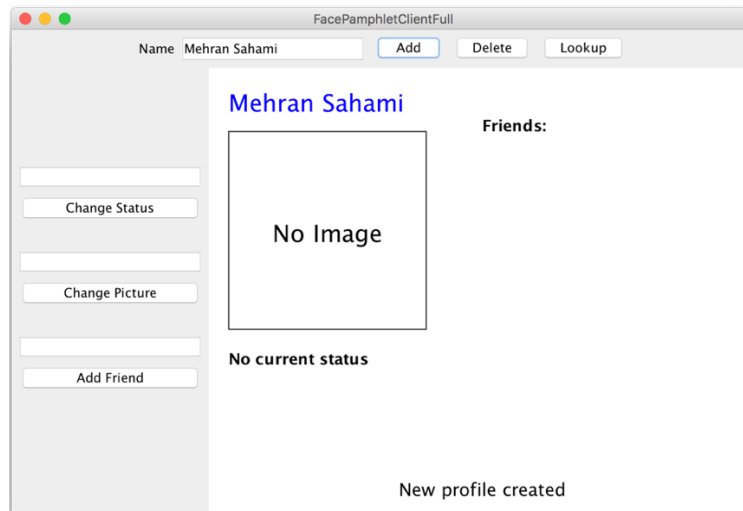
The SteamTunnel client program provides a simple interface for users to add, delete, and look up profiles in this social network. It talks to the server in order to display the right information to the user – for instance, when the user wants to pull up a certain user's profile, the client requests information about that profile (such as its status message, image, etc.) from the server.

As part of the starter files, we include for you demo JAR files with completed versions of both the client and server programs. These demos are helpful both in visualizing how users will interact with your client program (and thus why the server is important), and in testing your programs' functionality. You can run a JAR by double clicking on it.

First run the demo server. Then run the demo client...

Continued on next page

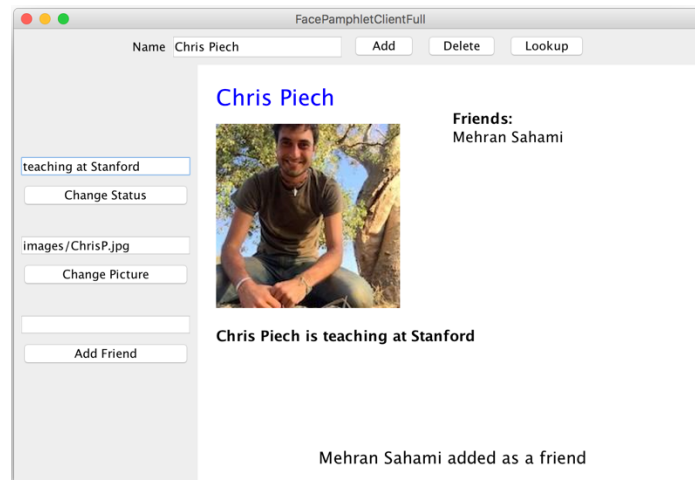
When the demo client application starts, it shows a blank screen with a few interactors. To create a new profile, the user would enter a name in a **Name** text field and click the **Add** button. For example, say we entered **Mehran Sahami** in the text field and clicked **Add**. Assuming there is not already a profile with the name "Mehran Sahami" in the social network, we would create a new profile for Mehran which is then displayed on the client. Importantly, the Steam Tunnel data lives on the server – so at this point the client will be communicating to your server program that a new user should be created. The client will then ask the server about the user's status and friends. Based on the server's response, the client displays the profile:



In this profile displayed above, we note five display elements of interest:

- **Name:** The name associated with the profile ("Mehran Sahami") is displayed prominently in the top left corner of the display canvas.
- **Image:** Although there is currently no image associated with this profile, there is space available to display a picture immediately under the name of the profile.
- **Status:** Under the area for the image, the *current status* of the person with this profile is displayed. Since a newly created profile does not have a status yet set, the display simply shows the text "No current status".
- **Friends:** To the right of the profile's name, there is the header text "Friends:", and space available under this text to list the friends of this profile. Again, since we have just created a new profile, there are no friends yet associated with it, so there are no entries listed under the "Friends:" header.
- **Application Message:** Centered near the bottom of the display canvas is a message from the application ("New profile created") letting us know that a new profile was just created (which is the profile currently being displayed).

The user can use the visible interactors to then edit information about the current profile. Here is a version of Chris's profile after (1) setting his picture to "images/ChrisP.jpg" (2) setting his status to "teaching at Stanford" and (3) having him become friends with Mehran Sahami (another CS106A instructor in the CS department in our social network):



In this way, you can use the client to continue building your social network. You can add more users, look up a user (and go to their profile) and edit their status, pictures, or add friends. Each of these steps relies on communication with your server.

NOTE: because images are being sent over the network, it may take a few seconds for profile pictures to be sent/received, both in the client and server programs. This is expected behavior.

The SteamTunnel Server

The heart and soul of this application is a program called a “server”. In theory you could execute the server on a computer very far away, though while testing you will be running it on your own computer (your computer can treat two programs as though they are communicating over the internet, even though they are actually on the same computer).

To explain why we need a server, first think: how would you feel if you went to [instagram.com](https://www.instagram.com) and created a profile, but then nobody in the world could see it, and the next time you opened up the website your profile was gone? In order for data to be visible across the internet, and to persist across client sessions (e.g. your phone and your computer), it is stored and managed by a separate program: the server.

The server stores all of the data and contains the logic for keeping track of profiles and getting and setting profile properties. It doesn’t display the data to a user. That is the job of a client. When the server receives a request (usually from a client), it updates its internal data and sends back a string response. Here is an example of a server that has received many requests (e.g. the **addProfile** command) and its corresponding responses:

```

FacePamphletServerSolution [completed]
Starting server on port 8000
Received request: containsProfile (name=Mehran Sahami)
Responding -> false
Received request: addProfile (name=Mehran Sahami)
Responding -> success
Received request: getImage (name=Mehran Sahami)
Responding -> 
Received request: getStatus (name=Mehran Sahami)
Responding -> 
Received request: getFriends (name=Mehran Sahami)
Responding -> []
Received request: setImage (name=Mehran Sahami,
imageStr=_9j_4AAQSkZJRgABAgAAQABAAD_2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAs
Responding -> success
Received request: getImage (name=Mehran Sahami)
Responding -> 
_9j_4AAQSkZJRgABAgAAQABAAD_2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8U
Received request: getStatus (name=Mehran Sahami)
Responding -> 
Received request: getFriends (name=Mehran Sahami)
Responding -> []
Received request: setStatus (name=Mehran Sahami, status=coding like
Responding -> success
Received request: getImage (name=Mehran Sahami)
Responding -> 
_9j_4AAQSkZJRgABAgAAQABAAD_2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8U
Received request: getStatus (name=Mehran Sahami)
Responding -> coding like a fiend
Received request: getFriends (name=Mehran Sahami)
Responding -> []
Received request: containsProfile (name=Julie Zelenski)
Responding -> false
Received request: addProfile (name=Julie Zelenski)
Responding -> success
Received request: getImage (name=Julie Zelenski)
Responding -> 
Received request: getStatus (name=Julie Zelenski)
Responding -> 

```

Don't be fooled by its textual display. This server is much more than just a **ConsoleProgram**. It is the backend of this internet application.



Here's a way of thinking of a server: A server is a bit like a database (e.g. **NameSurferDataBase**), and a client is like the controller (e.g. **NameSurfer**).

The functionality of a server is handled almost entirely by a single method:

String requestMade(Request request)

This method receives a request that has a command, and optionally some parameters. The server processes the request, updates its database and returns a response as a **String**. We'll talk about what a **Request** is in the following section.

In order to support a **SteamTunnel** client, your server is going to need to handle nine different types of request commands:

Command	Parameters	Response
addProfile	name	Creates a profile with the given name. Returns "success" or, if the profile already exists, returns an error message.
containsProfile	name	Returns "true" if a profile with the given name exists, and "false" otherwise.

<code>deleteProfile</code>	<code>name</code>	Removes a profile from the database. Returns “success” or, if the profile doesn’t exist, returns an error message.
<code>setStatus</code>	<code>name, status</code>	Sets the status of the user with the given name. Returns “success” or, if the profile doesn’t exist, returns an error message.
<code>getStatus</code>	<code>name</code>	Returns the status of the user with the given name, or the empty string if the user exists but does not have a status. Returns an error message if the profile doesn’t exist.
<code>setImage</code>	<code>name, imageString</code>	Sets the image for the user with the given name. Return “success” or, if the profile doesn’t exist, returns an error message.
<code>getImage</code>	<code>name</code>	Returns the profile image of the user with the given name, or the empty string if the user exists but does not have an image. Returns an error message if the profile doesn’t exist.
<code>addFriend</code>	<code>name1, name2</code>	Makes the user with name <code>name1</code> friends with user with name <code>name2</code> , and vice versa. Returns “success”, or an error message if either a) either user does not exist, b) if they are already friends, or c) if they are the same person.
<code>getFriends</code>	<code>name</code>	Returns the list of friends, as a string, for the user with the given name. Returns an error message if the profile doesn’t exist.

The SteamTunnel server program consists of several separate class files, as follows:

- **SteamTunnelServer**—This is the main program class that runs the server. It is a **ConsoleProgram** (though this is just for viewing what the server is doing – the user does not interact with it) and it is responsible for responding to requests from the client.
- **SteamTunnelProfile**—This class should encapsulate all the information for a single profile in the social network. Given a **SteamTunnelProfile** object, you can find out that profile's name, associated image (or lack thereof), associated status (or lack thereof), and the list of names of friends for that profile.
- **ServerTester** —This program is already written for you. It will send test requests to your server to help you test your server’s functionality.

To help you develop your program in stages, we have outlined some development milestones below.

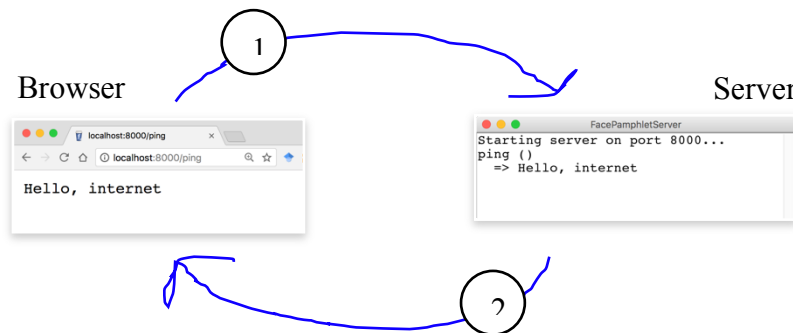
Milestone 1: Ping

Let’s start out with a simple task: write code to have your server respond to the “ping” command. If your server receives a request with the command “ping”, you should simply return the string “Hello, internet”. Open **SteamTunnelServer** to get started.

The starter code is already set up with a **SimpleServer** instance variable. In the **run** method when we call **start()** on the server variable, this signifies your program saying it is ready to receive incoming requests. Every time a request is sent to your program, the method **requestMade** will be called with the details of the request. As we talked about in class there are two methods that you can call on the request that you are passed in: **getCommand()**, which returns the request’s command, and **getParam(key)** which returns the value associated with a request parameter. Your task for this milestone is to update the **requestMade** method to check if a given request has the command “**ping**”, and if so return “**Hello, internet**”.

For now, we can test out our ping response by making a request to your server from a web browser (e.g. Chrome or Firefox). Run your server program and navigate to <http://localhost:8000/ping> to send a request to the server with the command “ping”. Your browser will display the string that your server sends back.

The browser sends a request to the server with the command “ping”






The server responds with the string “Hello, internet” which the browser displays

The server in the picture above **printlns** the received request and the response returned. You do not have to imitate this functionality—what matters is that your server **returns** the appropriate string. Having said that, console output is useful for debugging.

You now have a program that is receiving (and responding to) an internet request! Next up, let’s figure out how to store the data for SteamTunnel, and then we can make our server process commands relevant to the SteamTunnel client.

Milestone 2: Implement the `SteamTunnelProfile` class

The server is responsible for storing all of `SteamTunnel`'s data so that it can respond to requests from the client. We are not going to explicitly tell you how to structure your data. Instead, on a high level, it is useful to think of the server as keeping track of many profiles, each of which the server can look up based on the user's name:

"Chris Piech"	→	<pre> name = "Chris Piech" status = "teaching" friends = ["Mehran Sahami", "Julie Zelenski"] image =  </pre>
"Mehran Sahami"	→	<pre> name = "Mehran Sahami" status = "teaching" friends = ["Chris Piech", "Julie Zelenski"] image =  </pre>
"Julie Zelenski"	→	<pre> name = "Julie Zelenski" status = "" friends = ["Mehran Sahami", "Chris Piech"] image =  </pre>

We have provided the shell of a class `Profile`. This class defines a new variable type which represents one user's profile.

The `Profile` class encapsulates the information pertaining to one profile in the social network. That information consists of four parts:

1. The name of the person with this profile, such as "`Chris Piech`"
2. The status associated with this profile. This is just a `string` indicating what the person associated with the profile is currently doing. It should initially be the empty string.
3. The image associated with that profile. There should initially be no image.
4. The list of friends of this profile. The list of friends is simply a list of the *names* (`strings`) that are friends with this profile. This list starts empty. The data structure you use to for this is left up to you.

Fill in the **Profile** class such that it is a fully functional variable type. You may implement any methods that you like. We suggest the following (and have included stubs for the methods in the starter code):

```
public Profile(String name)
```

In this constructor, initialize the state of a new profile with the given name.

```
public String getName()
```

In this method, return the name associated with this profile.

```
public GImage getImage()
```

In this method, return this profile's image, or null if this profile has no image.

```
public void setImage(GImage image)
```

In this method, set the image associated with this profile to be the provided image.

```
public String getStatus()
```

In this method, return the status associated with the profile, or the empty string if this profile has no status.

```
public void setStatus(String status)
```

In this method, set the status associated with the profile to be the provided status.

```
public boolean addFriend(String friend)
```

In this method, add the given friend name to this profile's list of friends if they are not already friends. If the name is not already in this profile's friends list, it adds it and returns **true**. If the name is already in this profile's friends list, it does not modify the friends list, and returns **false**.

```
public boolean removeFriend(String friend)
```

In this method, remove the given friend name from this profile's list of friends, if they are friends. If the name is in this profile's friends list, it removes it and returns **true**. If the name is not in this profile's friends list, it does not modify the friends list, and returns **false**.

```
public ArrayList<String> getFriends()
```

In this method, return a list of this profile's friend names.

Milestone 3: Design a datastructure to store your data

Think about what data structure you might use to store all social network data. You will need to quickly access and modify a collection of Profiles.

Milestone 4: Handle `addProfile`, `containsProfile` and `deleteProfile`

Next implement the server functionality to handle requests related to managing profiles. These may be sent from the client if, for instance, the user hits the “Add” button to add a new profile (which would send an `addProfile` request). The relevant commands, which are listed earlier in the handout, are repeated here for convenience:

Command	Parameters	Response
<code>addProfile</code>	<code>name</code>	Creates a profile with the given name. Returns “success” or, if the profile already exists, returns an error message.
<code>containsProfile</code>	<code>name</code>	Returns “true” if a profile with the given name exists, and “false” otherwise.
<code>deleteProfile</code>	<code>name</code>	Removes a profile from the database. Returns “success” or, if the profile doesn’t exist, returns an error message.

A request is a lot like the client program trying to execute a method on your server. The command is akin to the method name, and just like a method call, requests can contain parameters.

Start by modifying the `requestMade` method so that it can handle requests with the command `addProfile`. When your server receives such a request, the server should prepare to generate a new profile for a new user. Every request with command `addProfile` will include a parameter with key “name”. You can get the value of this parameter by calling:

```
request.getParam("name"); // the name associated with this request.
```

If your SteamTunnel server does not already have a profile with the given name, create a new profile, store it, and return the string “success”.

If your SteamTunnel server already has a profile with the given name, you should return an error message. An error message is any string which starts with “Error:”. The remainder of the string describes what went wrong. For example:

```
"Error: Database already contains a profile with name Trogdor"
```

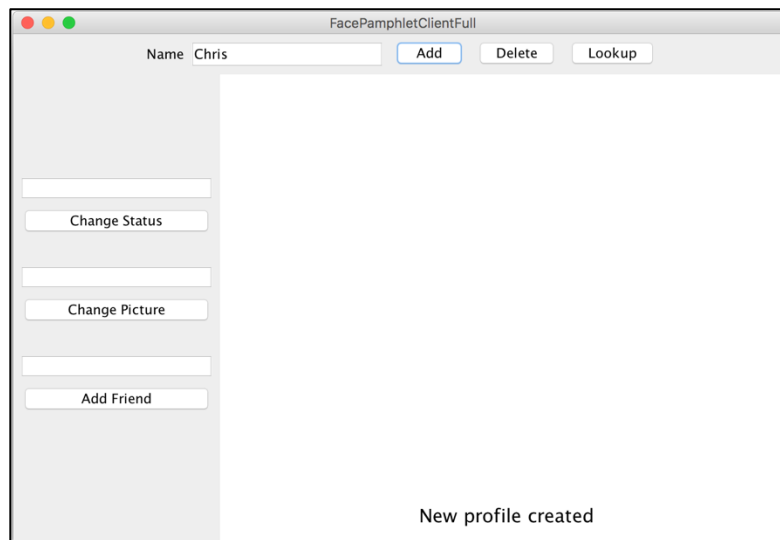
For this assignment, we don’t mind what description you send, as long as the string you return is an error message starting with “Error:”. For `addProfile`, you do not need to do any further error handling. For instance, you do not need to worry about correctly responding to a command that does not have a “name” parameter.

Next, you should expand your `requestMade` method to respond to the commands `containsProfile` and `deleteProfile`. Your implementation should match the specifications in the table above.

Testing the Server

Servers need to be tested thoroughly. In your project we provide a program called `ServerTester`. This program will send test requests to your server and let you know if the responses were the ones we expected. Note that this tester is *not exhaustive* and may not alert you to all errors. We highly recommend doing additional testing as you implement different commands. First, launch your `SteamTunnelServer` program, then launch the included `SteamTunnelServerTests`. You should pass all of the tests with the commands you just implemented.

At this point, if you start your server and then execute the provided demo client by double-clicking on it, you will get a client that can add, delete and lookup a profile, but since the client can't get the status, image or friends from the server, it can't display a profile:



Milestone 4: Handle requests to get and set the status and profile image

Want that client program to be able to do more? You need a server that can handle more requests! Now that the server can store users, the next step is to expand the `requestMade` method so that it can handle requests to get and set a user's status, and to get and set a user's image. The relevant commands, which are listed earlier in the handout, are repeated here for convenience:

Command	Parameters	Response
<code>setStatus</code>	<code>name, status</code>	Sets the status of the user with the given name. Returns “success” or, if the profile doesn’t exist, returns an error message.
<code>getStatus</code>	<code>name</code>	Returns the status of the user with the given name, or the empty string if the user exists but does not have a status. Returns an error message if the profile doesn’t exist.
<code>setImage</code>	<code>name, imageString</code>	Sets the image for the user with the given name. Return “success” or, if the profile doesn’t exist, returns an error message.
<code>getImage</code>	<code>name</code>	Returns the profile image of the user with the given name, or the empty string if the user exists but does not have an image. Returns an error message if the profile doesn’t exist.

For all four requests, the parameter “name” tells you which user’s data should be modified. If name refers to a user who does not exist, your server should return an error message. Again, it does not matter what description you provide in your error message, as long as the string starts with “Error:”. You do not need to do any further error checks.

Two commands in this milestone have to do with getting and setting images. When sending information over the internet, *everything* has to be text; even images. For this reason, when handling images in your server, you will need to convert between `GImage` and `String`. We have provided two methods to help you do this:

```
/* Converts a GImage to its string representation
String SimpleServer.imageToString(GImage image)

/* Converts a string representation of an image to a GImage
GImage SimpleServer.stringToImage(String str)
```

For instance, when you receive a request with command `setImage`, the parameter `imageString` is a string, not a `GImage`. To convert it to a `GImage`, you could write:

```
String imageString = request.getParam("imageString");
GImage image = SimpleServer.stringToImage(imageString);
```

Similarly, when you receive a request with command `getImage`, to convert a `GImage` on the server to a `String` to send it as a response, you could say:

```
GImage image = ...
String imageString = SimpleServer.imageToString(image);
```

Note: a null image converts to the empty string, and vice versa. Calling `stringToImage` with a poorly formatted input string throws an `IllegalArgumentException`.

Milestone 5: Handle requests to `addFriend` and `getFriends`

Everything is better with friends. Your final task is to expand your `requestMade` method to also handle requests to `addFriend` or `getFriends`. The relevant commands, which are listed earlier in the handout, are repeated here for convenience:

Command	Parameters	Response
<code>addFriend</code>	<code>name1, name2</code>	Makes the user with name <code>name1</code> friends with user with name <code>name2</code> , and vice versa. Returns “success”, or an error message if either a) either user does not exist, b) if they are already friends, or c) if they are the same person.
<code>getFriends</code>	<code>name</code>	Returns the list of friends, as a string, for the user with the given name. Returns an error message if the profile doesn’t exist.

When adding a friend, the request will contain two parameters: `name1` and `name2`. If the two parameters are names of two different users in the database who were not previously friends you should update your database so that the users are now friends. Friendship in SteamTunnel is reciprocal, so if Chris becomes friends with Julia, then Julia becomes friends with Chris. Once the users are made to be friends, the server responds with the string “success.”

When the server receives a request with the command `getFriends`, return the user’s list of friends as a string. For example, if Chris is friends with Julia and Laura, then when the server receives a `getFriends` request with the parameter `name` = Chris, return:

```
"[Julia, Laura]"
```

which is a string representation of the friend list. If a user has no friends, simply return the string:

```
"[]"
```

Hint: `ArrayLists` have a method `toString` which returns exactly this string representation.

The following cases should lead you to return an error message:

- Either `name1` or `name2` are users who are not in the database.
- The two users are already friends.
- Both `name1` and `name2` are the same. Eg `name1` = Chris and `name2` = Chris. Sorry Chris, you can’t be your own friend ☹.

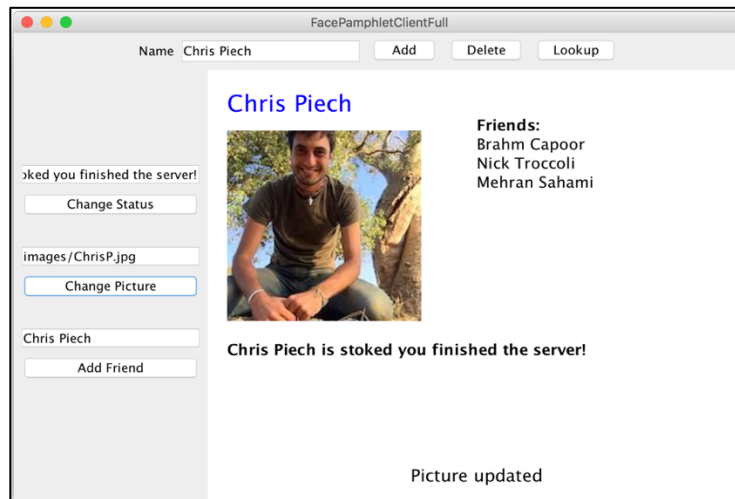
In all cases where you should return an error, make no changes to the database. Again, it does not matter what description you provide in your error message, as long as the string starts with “Error:”. You do not need to do any further error checks.

Finishing Up The Server

At this point, when you launch the `SteamTunnelServerTests` you should pass (almost) all of the tests. One request command that you may have to revisit is the `deleteProfile` command. When you delete a profile, because friendships are reciprocal, you should remove that person's name *from all their friends' friend lists* as well. For example, if Chris and Mehran are friends and then later Mehran's profile is deleted, Mehran should no longer be listed in Chris' friends. Sad. That's life in the city.

One other point is that, if you receive a request with an *unknown* command, you should simply respond with an error message.

At this point, you have implemented all required server functionality! If you run your server and then run the provided demo client, the client and your server should start to communicate and the result should be a fully functional internet application:



How can other computers access my server?

Right now your server is running on <http://localhost:8000> which is a special address that means “my computer port 8000”. For a client running on another computer to access it you first need to get a public web-address for your computer. You can use a service like [ngrok](#) to give you a temporary address that will map to your computer (in a way that is respectful of Stanford's firewalls).

Extensions

Extension: Implementing the SteamTunnel client

If you finish early you can implement the SteamTunnel client as an extension. The details are included in the client handout.

Extension: Federated Network

Can you come up with a system where your server can communicate with another SteamTunnel written by another student in CS106A?