Chris Piech
CS 106A

Handout #32
March 8, 2017

Assignment #7—FacePamphlet
Due: 5pm on Friday, March 17th

Note: No late days (free or otherwise) may be used on Assignment #7

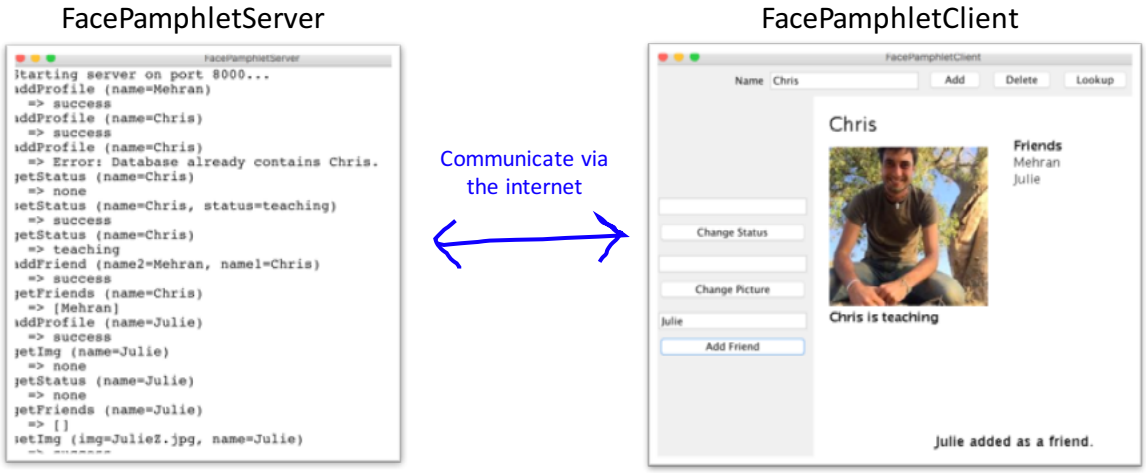
Your Early Assignment Help hours: 7:00pm-8:00pm, Thurs, March 9th in Hewlett 200

Based on an assignment (with the same name) by Mehran Sahami.

S*ocial networking* applications (such as Facebook, LinkedIn, and Snapchat) are used by billions of people. They are immensely popular particularly because of the power of the internet to connect people. In this vein, your job for this assignment is to create a application that keeps track of a simple, internet based, social network: FacePamphlet.

Internet Application

Applications like FacePamphlet, that interact over the internet, are actually two separate programs: the "server" (aka the cloud, aka the backend) and the "client" (or frontend). The server is the program that stores all of the data of your social network. The client is the program that runs on a user's computer or phone (did you know Facebook's datacenters and android phones both run Java?).



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 database and returns strings back to the client. All of these communications are sent over the internet.

As a note: this is the first time ever that CS106A students are writing a server that interfaces with a client over the internet (Previous versions of FacePamphlet were quite different). Woot. We have done everything we can to make this assignment an experience that maximizes education while still being respectful of your time. We get that it's the end of the quarter. In this assignment you are going to write **the server** (or backend) of the FacePamphlet application. You will practice the essential concepts that we have

learned in the second half of the course, and in addition learn the crucial parts of how to code for the internet. You can, optionally, write the client as extra credit :-).

Before you read any further, if you haven't seen the lecture from Wed **March 8th**, you may want to go check it out. In that lecture we go over concepts directly related to this assignment.

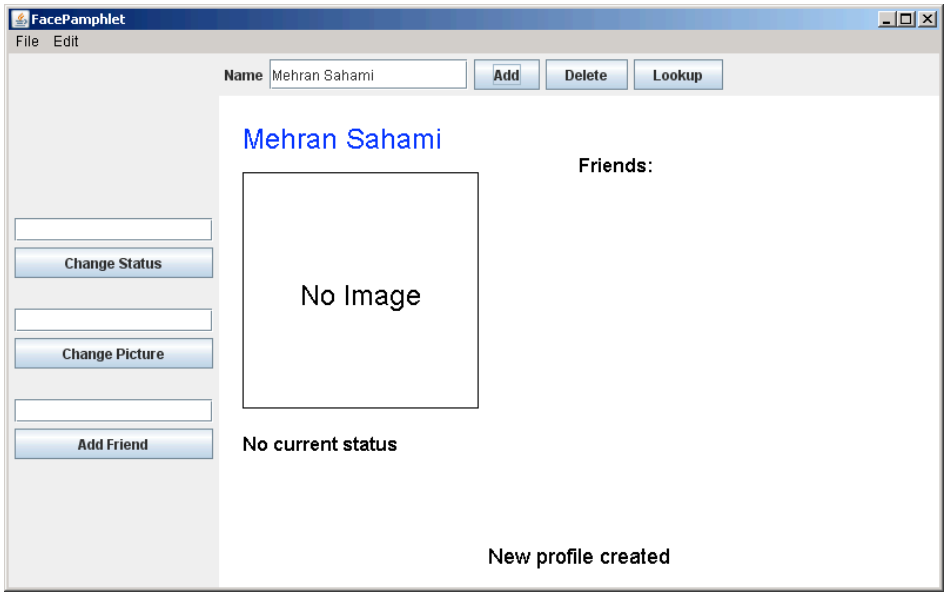
The Assignment

For this assignment, you will create a server program that keeps tracks of the information in the simple social network called FacePamphlet. Your server will manage all user's profiles. A profile includes: the person's name associated with that profile, an optional image file name for a profile picture, an optional "current status" for the profile (which is basically just a `String` indicating what activity the owner of that profile is currently engaged in), and a list of friends for each profile.

A Walkthrough of the Client Program

Your job in this assignment is to write the server (**writing the client is optional**). However, in order to understand *why* the server is important it is useful to first walkthrough how humans will interact with the client program. Initially, let assume the social network starts out empty (i.e., it contains no profiles). When the client application starts it shows a blanks screen with a few interactors.

To create a new profile, the user would enter a name in a **`Name`** text field and click an **`Add`** button. For example, say we entered **Mehran Sahami** in the text field and clicked **`Add`**. Since there is not already a profile with the name "Mehran Sahami" in the network, we would create a new profile for Mehran which is then displayed on the client. Importantly, the FacePamphlet data lives on the server – so at this point the client will be communicating to your program that a new user has just been created. The client will then ask the server about the user's status and friends. Based on the servers 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).

Updating the Profile

Both the user's status and profile picture can be updated using the client interactors. Here is a version of Mehran's profile after (1) setting his picture to MehranS.jpg (2) setting his status to "coding like a fiend" and (3) having him become friends with Julie Zelinski (another intrepid lecturer in the CS department):

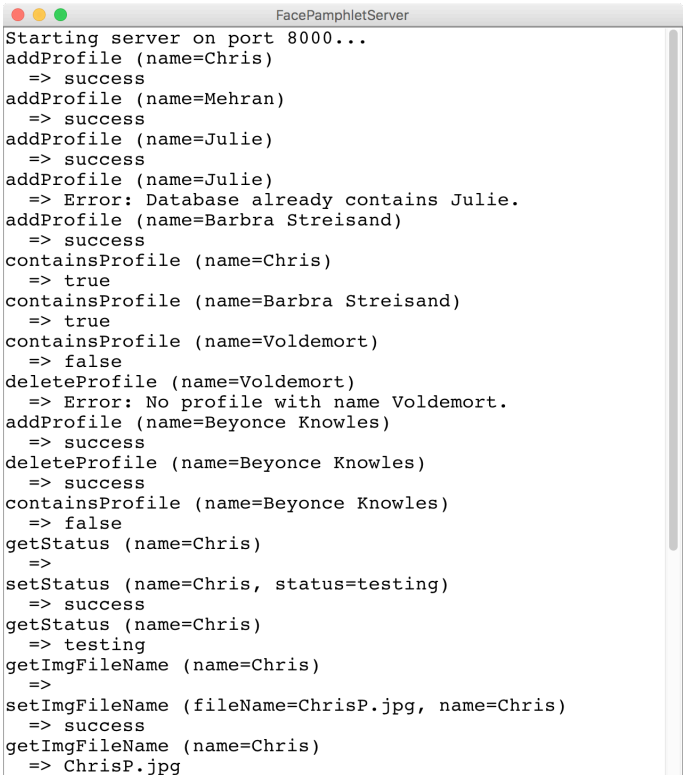
With the client you can continue building the social network. You can add more users, lookup a user (and go to their profile) and edit their status, pictures or add friends. Each of these steps will require communication with your server.

A Walkthrough of the Server Program

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 laptop.

To explain why we need a server first think: How would you feel if you went to facebook.com, created a profile, but then nobody in the world could see it, and moreover the next time you open up the website your profile was gone? In order for data to be visible across the internet, and to persist it is stored and managed by a separate program: the server.

The server stores all of the data and contains the logic for creating, deleting 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 (which often comes from the client), it updates its internal data and sends back a string. Here is an example of a server that has received many requests (eg **`addProfile`**) and its corresponding responses:



Don't be fooled by its textual display. This server is much more than just a **`ConsoleProgram`**. It is the backend of your first (presumably) internet application.

Here's a way of thinking of a server: A server is a bit like we took a database class (eg NameSurferDatabase) and turned it into *its own program* that we call a server. What used to be the cavas (eg NameSurferCanvas) becomes the client.

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

```
String requestMade(Request request)
```

This method recieves a request that has a command, and optionally some parameters. The server processes the request, updates its database and returns a respose as a String.

In order to support a FacePamphlet application, your server is going to need to handle nine different request commands.

Command	Parameters	Response
`addProfile`	`name`	Creates a profile with the given name. Return "success" or, if the profile already exists, returns an error message.
`containsProfile`	`name`	Checks if a profile with the given name exists. Returns "true" or "false."
`deleteProfile`	`name`	Removes a profile from the database. Returns "success" or, if the profile doesn't exist, returns an error message.
`getStatus`	`name`	Returns the status of the user with the given name (or ""). Returns an error message if the profile doesn't exist.
`getImgFileName`	`name`	Returns the image fileName of the user with the given name (or ""). Returns an error message if the profile doesn't exist.
`getFriends`	`name`	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.
`setStatus`	`name, status`	Sets the status of the user with the given name. Returns "success" or, if the profile doesn't exist, returns an error message.
`setImgFileName`	`name, fileName`	Sets the image fileName for the user with the given name. Return "success" or, if the profile doesn't exist, returns an error message.
`addFriend`	`name1, name2`	Makes user with name1 friends with user with name2 and vice versa. Return "success", or an error message if either user does not exist, or if they are already friends.

Time to get started!

Similar to the NameSurfer assignment, the FacePamphlet program is broken down into several separate class files, as follows:

- **`FacePamphletServer`**—This is the main program class that runs the server. It is a **`ConsoleProgram`** and it is responsible for responding to requests from the client.
- **`FacePamphletProfile`**—This class should encapsulate all the information for a single profile in the social network. Given a **`FacePamphletProfile`** 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 at your server and let you know what is working and what is not.

To help you with regard to developing your program in stages, we outline some development milestones below, along with more details regarding implementing the functionality provided in the program.

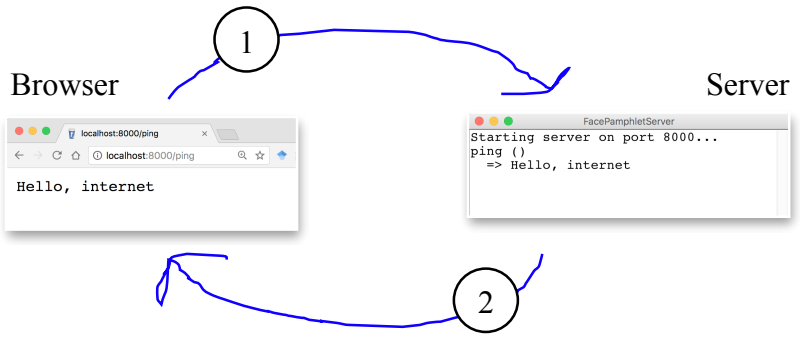
Milestone 1: Ping

Before we program requests that support the FacePamphlet client, let's start out with a simple task: write code to have your server respond to a ping command. If your server receives a request with command "ping" you simply return the string "hello, internet". Open **`FacePamphletServer`** 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 your program says it is ready to receive incoming requests. Every time a request is sent to your computer 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. Update the **`requestMade`** method to check if a given request has command equal to **`"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 (eg Chrome or Firefox). Start your server 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 command "ping"



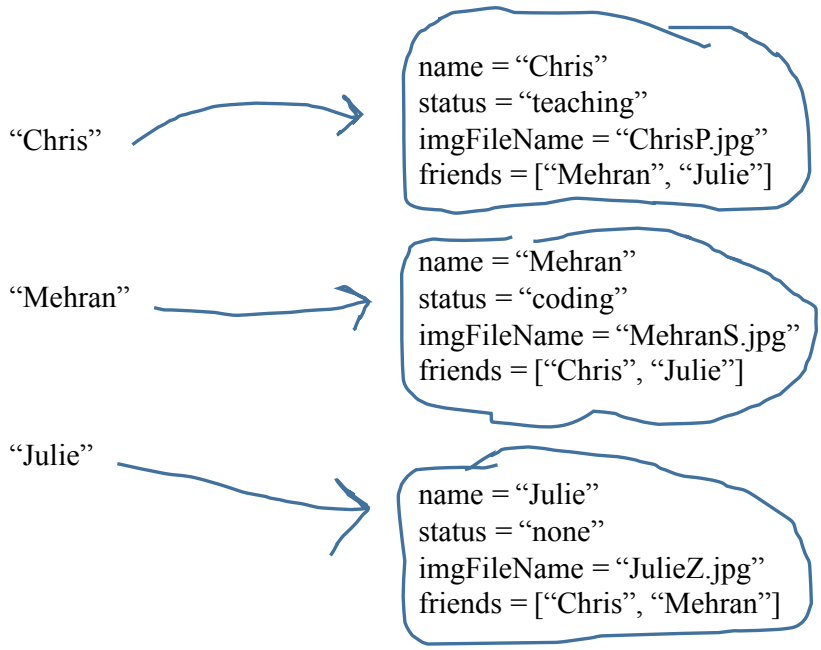
The server response with the string "Hello, internet" which the browser displays

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

Call your loved ones! You now have a program that is recieving (and responding to) an internet request. Next up, let's make our server process commands relevant to FacePamphlet.

Milestone 2: Implement the `FacePamphletProfile` class

The server is responsible for storing all of FacePamphlet'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:



We have provided the shell of a class **`FacePamphletProfile.`** This class defines a new variable type which represents one user's profile. The starter file for the **`FacePamphletProfile`** class includes definitions for all of the public methods we expect you to define. The method definitions in the starter files, however, do nothing useful (they are just ***stubs***), although they occasionally include a **`return`** statement that gives back a default value of the required type. For example, the **`getName`** method always returns the empty string ("") to satisfy the requirement that the method returns an **`String`** as defined in its header line.

The **`FacePamphletProfile`** 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 **`"Mehran Sahami"`** or **`"Julie Zelenski"`**

2. The status associated with this profile. This is just a **`String`** indicating what the person associated with the profile is currently doing. Until it is explicitly set, the status should initially be the empty string.
3. The image fileName associated with that profile. This is a **`String`**. Until it is explicitly set, this field should initially be the empty string.
4. The list of friends of this profile. The list of friends is simply a list of the *names* (i.e., list of **`String`**s) that are friends with this profile. This list starts empty. The data structure you use to keep track of this list is left up to you.

Fill in the **`FacePamphletProfile`** class such that it is a fully functional variable type.

Now that you have a new variable type (**`FacePamphletProfile`**) what data structure can you use that will enable you to look up a user's profile based on their name?

Milestone 3: Handle requests to addProfile, containsProfile and deleteProfile

When a person using a client application hits the add button, the client is going to send a message to the server to create a new profile. You must implement the server to receive that request.

The first set of FacePamphlet commands for you to implement are ones that tell the server to create or delete a profile, and to return whether or not a profile exists. The commands, which were listed above, are repeated here for convenience.

Command	Parameters	Response
`addProfile`	`name`	Creates a profile with the given name. Return "success" or, if the profile already exists, returns an error message.
`containsProfile`	`name`	Checks if a profile with the given name exists. Returns "true" or "false."
`deleteProfile`	`name`	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 FacePamphlet server does not already have a profile with the given name, create a new profile, store it, and return the string **`"success"`**.

If your FacePamphlet 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 the string:

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

Is an error message. For this assignment, when an error should be retured we don't mind what description you send, as long as the string you return is an error message.

For **`addProfile`** You do not need to do any further error handling. Specifically you do not need to worry about correctly responding to a command that does not have a **`"name"`** parameter. If you are having trouble understanding how to process a request, look at the example **`ChatServer`** from class on March 8th.

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

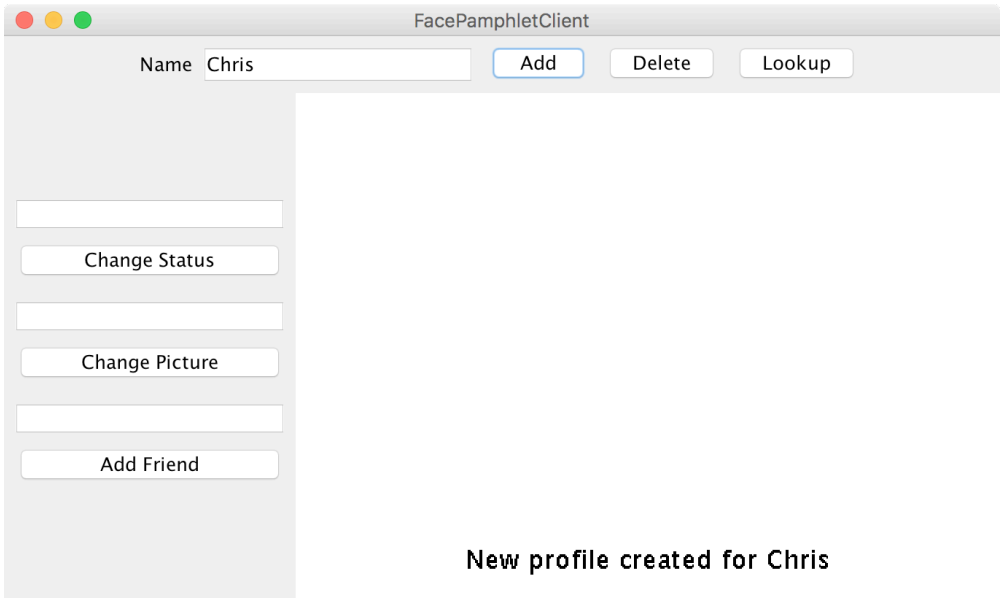
Testing the Server

Servers need to be tested thoroughly. In your project we provide a program called **`ServerTester`**. This program will send test requests at your server and let you know if the responses were the ones we expected. First, launch your **`FacePamphletServer`** program, then launch the **`ServerTester`**. You should pass all of the tests with the commands you just implemented.

In your project we also include an executable jar which contains a fully functional client:

```
FacePamphletClient.jar
```

The client is just a thin interface that needs a functional server . At this point, if you start your server and then execute **`FacePamphletClient.jar`** you will get a client that can add, delete and lookup a profile, but since the client can't get 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 that get and set a user's status, and to get and set a user's image file name.

`Command`	`Parameters`	Response
`getStatus`	`name`	Returns the status of the user with the given name (or ""). Returns an error message if the profile doesn't exist.
`setStatus`	`name, status`	Sets the status of the user with the given name. Returns "success" or, if the profile doesn't exist, returns an error message.
`getImgFileName`	`name`	Returns the image fileName of the user with the given name (or ""). Returns an error message if the profile doesn't exist.
`setImgFileName`	`name, fileName`	Sets the image fileName for the user with the given name. Return "success" or, if the profile doesn't exist, returns an error message.

For both status and image file name, if the user has yet to set a status or an image file name, a call to **`getStatus`** or **`getImgFileName`** should simply produce the empty string. Otherwise your job is to return the most recently set status or file name.

When the server receives a message like **`setStatus`** or **`setImgFileName`**, its job is to update its data to record that the user has a new status or image file name. If everything goes well the server simply returns the string "success" to indicate that the request was received and the information the request contained was saved.

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.

You may be wondering: hey why do we just save the image file name. What about the actual image? The SimpleServer class that we are using already knows how to receive an image file and save it. Thus your are only responsible for keeping track of the file name.

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.

Command	Parameters	Response
`getFriends`	`name`	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.
`addFriend`	`name1, name2`	Makes user with name1 friends with user with name2 and vice versa. Return "success", or an error message if either user does not exist, if they are already friends or if both names are the same.

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 FacePamphlet is reciprocal so if Chris becomes friends with Nick, then Nick becomes friends with Chris. Once the users are made to be friends, the server responds with the string "success."

The following cases should lead you to return an error:

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

In all cases where you should return an error, make no changes to the database.

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

```
"[Nick, Laura]"
```

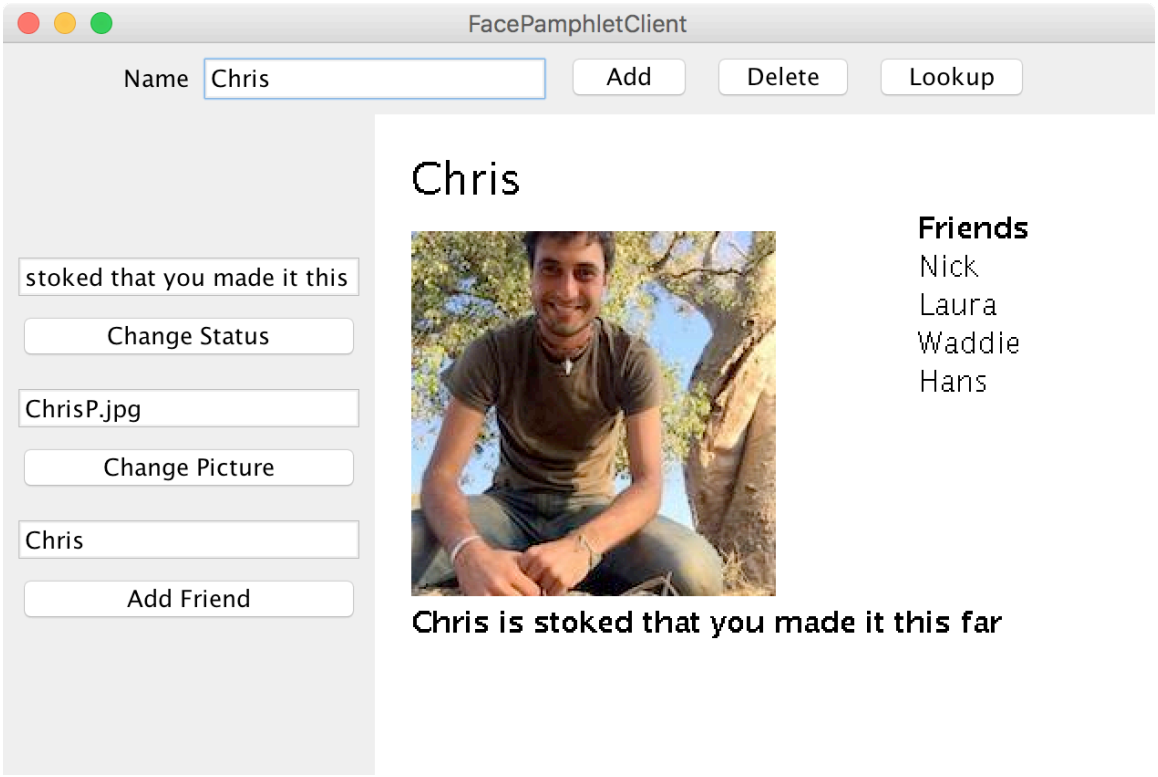
Which is a string representation of the friend list. Happily, **`ArrayLists`** have a method **`toString`** which returns exactly that string representation.

Finishing up

At this point, when you launch the **`ServerTester`** 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, you should remove that person's name from all friends lists. 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.

Congrats! If all of the tests from ServerTester are passing then you are done with the core functionality of FacePamphlet.

At this point, if you run your server and then run the **`FacePamphletClient.jar`**, the client and your server should start to communicate and the result should be a fully functional internet based application:



You are done with the required part of the assignment. So far we provided the client for you. If you want, you can write the client for extra credit.

Extra credit extension: Implement FacePamphletClient.

While it is pretty exciting to have created your own server, it would be amazing if you wrote the client too. Implementing the client requires you to use several of the features we have seen in class before (namely interactors and GObjects). You can find starter code for the client online [the starter code will be released by Friday March 10th].

The main new concept that you will have to implement if you are to embark on making a client, is how to generate a request and send it to your server. In the starter code we create a ping request and turn the result into a GLabel that we display in the center of the canvas. All server requests must be written in a try/catch block and sometimes you may want to handle exception gracefull and not crash.

Most of the client is self explanatory. You can observe the full functionality by running the FacePamphletClient.jar file in the server starter code. However, since this is an extension, you don't have to match our implementation exactly.

One piece of the client which is slightly non-obvious is how the client sends and loads actual image files to the server. When the client is told to change a user's profile image, the client does two things. It first sends the image file to the server. It then tells the server to set the imgFileName for the user. The SimpleClient class has two extra methods to send and receive image files:

```
/* save an image (with the given filename) to the server host */
String SimpleClient.saveImage(String host, String fileName)

/* load an image (with the given filename) from the server host */
GImage SimpleClient.getImage(String host, String fileName)
```

Both of these requests are processed automatically by the SimpleServer.

Implementing the client is not a small task, and completing the entire client implementation is worth substantial extra credit.

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 a nother 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).

Additional extension ideas

Here are some additional ideas for ways to extend your FacePamphlet program:

- *Keep track of additional information for each profile.* The current profile only keeps track of a name, image, status and a list of friends. In real social networks, there is much more information about users that is kept track of in profiles (e.g., age, gender, where they may have gone to school, etc.) Use your imagination. The more challenging issue will be how you appropriately display this additional information graphically in the profile display.
- *Click on friends to see their profile.* First of all, in your friends list you could keep a picture of each friend beside the name of the friend. Then, you could implement mouse listeners so that if the user clicks on one of the friends photos (or name) you go straight to that friend's profile.
- *Support for groups.* Many social networking applications allow for keeping track of "groups" (or "communities") that profiles can belong to. In many ways, being a member of a group is similar to having that group as a "friend"—a "group" has a list of members (similar to a list of friends for a profile) and each profile can be a member of many groups (much in the same way that a profile can have many friends). Adding support for groups would help make your social network more realistic and may not actually require too much work if you can leverage some of the conceptual similarities with respect to "groups" being like "friends".
- *Finding friends of friends.* Another interesting aspect of social networks is not only keeping track of how many people you have as friends, but also how quickly that

number grows as you consider all the friends of your friends, and their friends, and so on. Displaying these sorts of properties of the social network are neat features that show just how few degrees of separtion there are between people. Along these same lines, it would be interesting to find and display "friendship chains" that show the shortest sequence of friendship relations that create a chain from one profile to another. For example, if X is a friend of Y, and Y is a friend of Z, then a friendship chain exist that goes: X → Y → Z. Finding longer chains can be a fun and challenging problem.

- *Adjust the profile display as the application window is resized.* You got some practice with this already with the NameSurfer application and it would be an interesting extension to apply some of those same ideas here. The more challenging issue is how you would decide to change font sizes and the size of the image as the display size grew or shrank.
- *Go nuts!* There's really no shortage of ways that you could extend your FacePamphlet application. In fact, whole companies have been started based on creating a social network application with some cool new features. And if you do end up starting the next multi-billion dollar company based on social networking, just remember where it all started... CS106A!