CS 1410 Assignment 3: YouFace

In this assignment you will create a simple social networking website. It will run a web server on your computer where you can connect to it with your browser. The server will contact a central database (shared by all students in the course) to store and update information about users.

In the first part, you will extend and use your HTML classes to generate the three distinct screen types that make up the application. In the second part, you will write the server code that connects the browser to the database and implements the core logic of the application.

Part I

In the first part of this assignment, you will create a few class hierarchies for rendering the HTML pages that make up the application. When you have finished this part, you will have code to generate each of the three distinct screens that make up YouFace. Ultimately, there will be three classes used by the main application (written in part 2):

1. `LoginPage`: This class is instantiated without any arguments, and renders a complete page for the login screen. Example
2. `FeedPage`: Instances of this class are created with three arguments:
    - The name of the user currently logged in (a string)
    - A list of the most recent status updates of this user and his/her friends (a list of strings)
    - A list of the user's friends (a list of strings)

   The result is a complete rendering of the main page that a user interacts with, complete with a box for the user to update his/her status, a list of recent status updates, and a list of the user's friends. Example
3. `FriendPage`: Instances of this class are also created with three arguments:
    - The name of the friend (a string)
    - A list of the most recent status updates of the friend (a list of strings)
    - A list of all the user's friends (a list of strings)

   The resulting page shows the friend's most recent updates, and gives the user the option to remove this friend from his/her list. Example

The end goal of part 1 is to be able to create the necessary HTML pages as static documents. All live interaction with the database and web browser is reserved for part 2.

Put your work for part 1 in a file called `page.py`, and start it by importing your work from the HTML assignment:

```
from html import *
```

Make sure `html.py` is in the same directory.

HTML tags for forms

Start by creating a few more HTML tags used to create forms:

- `Form`: A form with fields to be filled out and submitted. Inherits from `BlockContainer`:

```
t = Form('/savedata')
# t renders as: '\n<form method="post" action="/savedata" />'
```

- `Label`: A label for form elements. Inherits from `Container`:

```
t = Label('targetelement')
# t renders as: '<label for="targetelement" />'
```

```
t.addText('Name:')
# t renders as: '<label for="targetelement">Name:</label>'
```

- `Input` should extend `Tag`. Forms usually require input fields, which have a type attribute and a name:

  ```
  t = Input('text', 'title')
  # t renders as: '<input type="text" name="title" />'
  ```

  Sometimes they also have values:

  ```
  t.addAttr('value', 'You Face')
  # t now renders as: '<input type="text" name="title" value="You Face" />'
  ```

Boxes

Anything displayed in the main section of the site is called a Box. In the examples shown above, these are the large items on the left of the page.

A Box is just a `div` tag with the class `box`, an `h1` title tag, and whatever contents are necessary for the specific kind of box being considered. The stylesheet ensures that it is rendered correctly on the page.

Start by creating the `Box` class. It inherits from the `Div` class in `html.py`. It is used as follows:

```
t = Box('Login')
# t renders as: '\n<div class="box">\n<h1>Login</h1>\n</div>'
```

Note that `Box` is just an ordinary `Div` with the `box` class, and a single `h1` tag inserted as its first contained element.

Inherit from this class four times to create the following classes:

1. `StatusBox`: This box takes the user's name as the sole argument to its constructor, and uses it to form the title of the box (passed to the superclass constructor). It then creates a form with specific elements, and adds the form as its child element.

   A `StatusBox` gives the user a place to enter a new status and click on a button labeled "Change" to save the update. Look at the box that says "Welcome, Alice" in the Feed page example above.

   By now you are well versed in creating tags, and should be able to combine them to create each of these elements based on an example. Your goal is to reproduce the following:

   ```
   t = StatusBox('Alice')


   # renders as:
   #
   # <div class="box">
   # <h1>Welcome, Alice</h1>
   # <form method="post" action="/status">
   # <p><label for="status">Change your status:</label><input type="text" name="status" /></p>
   # <p><input type="submit" name="change" value="Change" /></p>
   # </form>
   # </div>
   ```

   Note that the `div` tag with its `box` class and the `<h1>` tag all come from the superclass. Only the `<form>` element and its contents come specifically from the `StatusBox` class. If you are unsure how to place everything in the right order, load an interactive shell and trying creating the form by hand. From a python shell, type `from page import *` to get started, then start creating tags.

2. `RecentActivityBox`: This box takes a list of strings representing status updates. It puts them as items in an unordered list, which is then added as a contained element.

   Example:

   ```
   t = RecentActivityBox(['eating lunch', 'doing homework'])


   # renders as:
   ```

```
#
# <div class="box">
# <h1>Recent status updates</h1>
# <ul>
# <li>eating lunch</li>
# <li>doing homework</li>
# </ul>
# </div>
```

This box is used on the Feed page (displaying the most recent updates for the user and his/her friends), and on the Friend page (showing only the updates for the friend). Look at the examples above.

3. `UnFriendBox`: This box takes the name of a friend and creates a form with a single button to let the user end the friendship (at least in YouFace terms).

   Example:

   ```
   t = UnFriendBox('Alice')


   # renders as:
   #
   # <div class="box">
   # <h1>You are currently friends with Alice</h1>
   # <p>
   # <form method="post" action="/unfriend"><input type="hidden" name="name" value="Alice" />
   # <p><input type="submit" name="unfriend" value="Unfriend" /></p></form>
   # </p>
   # </div>
   ```

   This box is displayed at the top of the main column on each friend page. Look at the Friend page example above.

4. `LoginBox`: This has the most complicated form in the entire project, but that is not saying too much. This is the main part of the login page:

   Example:

   ```
   t = LoginBox()


   # renders as:
   #
   # <div class="box">
   # <h1>Login</h1>
   # <form method="post" action="/login">
   # <p><label for="name">Name:</label><input type="text" name="name" /></p>
   # <p><label for="password">Password:</label><input type="password" name="password" /></p>
   # <p><input type="submit" name="type" value="Login" /><input type="submit" name="type"
     value="Create" /><input type="submit" name="type" value="Delete" /></p>
   # </form>
   # </div>


   # note: long lines have been wrapped for formatting purposes
   ```

   These forms lend themselves well to method chaining. Try to create code that is organized and as simple as possible.

   Look at the Login page example above to see how this looks in the browser.

Gadgets

Anything displayed in the sidebar is called a Gadget. In the example pages, the sidebar is the collectiong of Gadgets along the right side of the page.

A gadget is very similar to a Box, but it is affected by different stylesheet rules and it is placed in a different part of the page.

Start by creating the `Gadget` class. It inherits from the `Div` class in `html.py`. It is used as follows:

```
t = Gadget('Links')
# t renders as: '\n<div class="gadget">\n<h1>Links</h1>\n</div>'
```

Note that `Gadget` is just an ordinary `Div` with the `gadget` class, and a single `h1` tag inserted as its first contained element.

Inherit from this class three times to create the following classes. As before, an example should give you enough information to create the class. Be sure to try out each example and make sure you get identical results. Look at the example pages above to see how these gadgets should look in the browser.

1. `LinksGadget`: This gadget holds a list of links. The constructor takes a link of tuples as its input. Each tuple contains a link URL (the `href` attribute of an anchor tag), and the text of the link (the part the user clicks to follow the link).

   Example:

   ```
   links = [('http://cit.cs.dixie.edu/', 'CIT'), ('/loginscreen', 'Login')]
   t = LinksGadget(links)


   # renders as:
   #
   # <div class="gadget">
   # <h1>Links</h1>
   # <ul>
   # <li><a href="http://cit.cs.dixie.edu/">CIT</a></li>
   # <li><a href="/loginscreen">Login</a></li>
   # </ul>
   # </div>
   ```

2. `FriendsGadget`: This gadget lists all of your friends, each one linking to a page about that friend. In addition, the gadget has a small form allowing the user to type in the name of another person and click a button to add that person as a friend. Note that the link for each friend is simply `'/friend/'` plus the name of the friend.

   Example:

   ```
   friends = ['Alice', 'Bob', 'Charlie', 'Dorothy']
   t = FriendsGadget(friends)


   # renders as:
   #
   # <div class="gadget">
   # <h1>Friends</h1>
   # <ul>
   # <li><a href="/friend/Alice">Alice</a></li>
   # <li><a href="/friend/Bob">Bob</a></li>
   # <li><a href="/friend/Charlie">Charlie</a></li>
   # <li><a href="/friend/Dorothy">Dorothy</a></li>
   # </ul>
   # <form method="post" action="/addfriend">
   # <p><input type="text" name="name" /><input type="submit"
       name="addfriend" value="Add Friend" /></p>
   # </form>
   # </div>


   # note: a linebreak was added for formatting purposes
   ```

3. `LogoutGadget`: This gadget displays a small form that contains nothing but a logout button.

   Example:

   ```
   t = LogoutGadget()


   # renders as:
   #
   # <div class="gadget">
   # <h1>Logout</h1>
   # <form method="post" action="/logout">
   # <p><input type="submit" name="logout" value="Logout" /></p>
   # </form>
   # </div>
   ```

Pages

The main contents elements are the boxes and gadgets. The page classes serve mainly to organize the page and to assemble the appropriate mixture of gadgets and boxes.

The root class of this hierarchy (which does not inherit from anything) is the `Page` class. It organizes the overall page by putting together tags organized as follows:

- Html
  - Head
    - Meta
    - Title (with the title of the page)
    - Stylesheet (linking to the stylesheet)
  - Body
    - Div with ID `maincontainer`
      - Div with ID `header`
        - H level one with a link to `'/'` and the title of the page as the link text
        - H level two with the subtitle of the page
      - Div with ID `maincolumn`
      - Div with ID `sidebar`

At the top of your `page.py` file, define global variables with the constant values that configure your page:

- `TITLE =` the title of your website (probably `'YouFace'`)
- `SUBTITLE =` the subtitle of your website
- `STYLESHEET =` the URL of your stylesheet (probably `'youface.css'`)
- `LINKLIST =` the list of link tuples for your links gadget.

To test part one, you should set these values to match the ones assumed by the page mockups. Copy this code; you can customize it in part 2:

```
TITLE = 'YouFace'
SUBTITLE = "A billion dollars and it's yours!"
STYLESHEET = 'youface.css'
LINKLIST = [
    ('http://cit.cs.dixie.edu/cs/cs1410/', 'CS 1410'),
    ('http://new.dixie.edu/reg/syllabus/', 'College calendar'),
]
```

The `Page` class also needs to know how to add new boxes and gadgets, and to render the main `Html` tag. To do this you should store three of the tags as fields in the `Page` object: the `Html` tag that is the root of the document, the `Div` tag with ID `maincolumn` (where boxes are added), and the `Div` tag with ID `sidebar` (where gadgets are added).

Then, add the following methods:

- `addBox`: takes a complete `Box` instance as its argument and adds it to the `Div` for boxes.

- `addGadget`: takes a complete `Gadget` instance as its argument and adds it to the `Div` for gadgets.

- `__str__`: renders the `Html` tag and returns the resulting string. (Hint: if you stored this tag as `self._html`, then use `str(self._html)` to render it.

- `__repr__`: calls `self.__str__()` and returns whatever it returns.

Example:

```
t = Page()

# renders as:
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
# <html xmlns="http://www.w3.org/1999/xhtml">
# <head>
# <meta http-equiv="content-type" content="text/html; charset=utf-8" />
# <title>YouFace</title>
# <link rel="stylesheet" href="youface.css" type="text/css" />
# </head>
# <body>
# <div id="maincontainer">
# <div id="header">
# <h1><a href="/">YouFace</a></h1>
# <h2>A billion dollars and it's yours!</h2>
# </div>
# <div id="maincolumn" />
# <div id="sidebar" />
# </div>
# </body>
# </html>

lg = LogoutGadget()
t.addGadget(lg)

# now renders as:
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
# <html xmlns="http://www.w3.org/1999/xhtml">
# <head>
# <meta http-equiv="content-type" content="text/html; charset=utf-8" />
# <title>YouFace</title>
# <link rel="stylesheet" href="youface.css" type="text/css" />
# </head>
# <body>
# <div id="maincontainer">
# <div id="header">
# <h1><a href="/">YouFace</a></h1>
# <h2>A billion dollars and it's yours!</h2>
# </div>
# <div id="maincolumn" />
# <div id="sidebar">
# <div class="gadget">
# <h1>Logout</h1>
# <form method="post" action="/logout">
# <p><input type="submit" name="logout" value="Logout" /></p>
# </form>
# </div>
# </div>
# </div>
# </body>
# </html>

rab = RecentActivityBox(['eating lunch', 'doing homework'])
t.addBox(rab)

# now renders as:
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
# <html xmlns="http://www.w3.org/1999/xhtml">
# <head>
# <meta http-equiv="content-type" content="text/html; charset=utf-8" />
# <title>YouFace</title>
# <link rel="stylesheet" href="youface.css" type="text/css" />
# </head>
# <body>
# <div id="maincontainer">
# <div id="header">
# <h1><a href="/">YouFace</a></h1>
# <h2>A billion dollars and it's yours!</h2>
# </div>
# <div id="maincolumn">
# <div class="box">
# <h1>Recent status updates</h1>
# <ul>
# <li>eating lunch</li>
# <li>doing homework</li>
# </ul>
# </div>
# </div>
# <div id="sidebar">
# <div class="gadget">
# <h1>Logout</h1>
# <form method="post" action="/logout">
# <p><input type="submit" name="logout" value="Logout" /></p>
# </form>
# </div>
# </div>
# </div>
# </body>
# </html>
```

The rest of the page hierarchy works by putting everything together in a series of short classes:

- `LoginPage` (inherits from `Page`): Creates a `LoginBox` box and adds it, then creates a `LinksGadget` gadget (using the links defined as `LINKLIST`) and adds it.

  When instantiated and rendered, this class creates a complete login page.

- `UserPage` (inherits from `Page`): This is the base class for all pages rendered when a user is logged in. Its constructor takes a list of friends as its argument, and it creates and adds three gadgets:

  1. `LogoutGadget`
  2. `FriendsGadget' (using the list of friends passed to the `UserPage constructor)
  3. `LinksGadget` (using the list of links defined as `LINKLIST`).

- `FeedPage` (inherits from `UserPage`): This is the main page a user sees when logged in. The constructor accepts three arguments: the user's name, the list of status update strings, and the list of friends. It passes the list of friends to its superclass, and uses the other arguments to add two boxes:

  1. `StatusBox` (using the user's name)
  2. `RecentActivityBox` (using the list of status updates)

- `FriendPage` (inherits from `UserPage`): This is the page a user sees after clicking on the name of a friend. The constructor accepts three arguments: the friend's name, the list of status update strings for the friend, and the list of all of the user's friends. It passes the list of friends to its superclass, and uses the other arguments to add two boxes:

  1. `UnFriendBox` (using the name of the friend)
  2. `RecentActivityBox` (using the list of status updates for the friend).

Part I Pass-off

To pass off part 1, you should complete all of the classes described above. Put your `html.py` and `page.py` files in the same folder with the starter kit files. Run `page_test.py` and `page_passoff.py` and fix your errors until both of these programs verify that your solution is correct.

Demonstrate your code and show it to the lab assistant.

You can download the starter kit here:

- starter kit

Part II

To start the second part, you will need to download a web microframework called *Bottle*. It is a single file that should go in the same directory as your project, and can be downloaded here:

- bottle.py

For more information about Bottle, see its website:

- http://bottlepy.org/docs/stable/

Put all of your work for part 2 in a file called `youface.py`, and start it by importing the necessary parts of Bottle. You will also need to import some functionality from a library called `xmlrpclib` to connect to the central database, and of course you will need your `page.py` classes.

Finally, you will need to actually connect to the database and launch the web server framework. The following starter code will get you started:

```
from bottle import get, post, run, request, response
from bottle import redirect, debug, static_file
import xmlrpclib
from page import *

DB_ADDRESS = 'http://youface.cs.dixie.edu/'
DB_SERVER = None

def main():
    global DB_SERVER, DB_ADDRESS
    print 'Using YouFace server at', DB_ADDRESS
    DB_SERVER = xmlrpclib.ServerProxy(DB_ADDRESS, allow_none=True)
    debug(True)
    run(host='localhost', port=8080, reloader=True)

if __name__ == '__main__':
    main()
```

If you put this in `youface.py` and run it, it will start a simple web server running on your local machine and listening on port 8080. You can test this by opening a browser and connecting to `http://localhost:8080`.

The result should be a "404 Not Found" error page. If you get this error, you have succeeded. It means that the web server successfully received your request, but did not know what to do with it. You will get a different error if your browser was unable to connect to your web server at all.

Static files

Your web server runs as an application that sits and waits for requests from the browser. Whenever it receives a request, the Bottle framework inspects the request and decides how to respond to it. It uses the URL that was requested to decide which *handler* should be invoked. If no suitable handler is found for a given URL, it responds with an error code.

The simplest handler works for static files. The only static file you need to serve is the stylesheet, `youface.css`. When the browser requests it, your server should simply pass the file to the browser.

To do this, you need to write a function to serve the static file, and you need to tell Bottle when that function should be invoked. Each URL in your application will work in a similar way. For this one, the complete code is:

```
@get('/youface.css')
def stylesheet():
    return static_file('youface.css', root='./')
```

The `@get` line tells Bottle that requests for the given URL should be handled by the function that follows. Whenever the browser asks for `/youface.css`, Bottle will call the `stylesheet` function, and whatever it returns will be handed to the browser.

For the actual function, you use a built-in function from Bottle called `static_file` that is designed to serve static files. It is possible to configure it to serve every file in a directory or other, more flexible variations, but for our needs only a single file is ever served.

Add this to `youface.py` and run it again. Be sure to put it *before* the function `main` is called (the call to `main` should remain the very last statement in your file). This time, point your browser at this URL to test it:

- http://localhost:8080/youface.css

If everything is working right, you should see the stylesheet in your browser window.

Logging in

The next step is to make a dynamic page, i.e., one whose contents do not come from a static file, but are instead computed each time the page is loaded.

The process is similar to the one for static files. Start by creating a `@get` command to telling Bottle which URL you are implementing. In this case, it is `/loginscreen`. Start with this code:

```
@get('/loginscreen')
def loginscreen():
    return 'Hello, world!'
```

Note the pattern: you give in a URL, then a function that should be called whenever the browser requests that URL. The job of the function is to form a web page and return it as a string.

Note that Bottle is set up to watch for changes in your `youface.py` file. If you made all the changes and then saved the file, it will automatically reload the file. You can start it running in one window, and edit it in a different window. If you save a change that causes it to crash, you will have to restart it manually, but otherwise it will automatically restart every time you make a change.

Test this new code by pointing your browser to the new URL:

- http://localhost:8080/loginscreen

You should see the text “Hello, world!” displayed in your browser. Try changing the message in the `loginscreen` function, save the result, wait for the server to reload, and then refresh the browser screen. You should see your new message.

Before you move on, you need to make one small change to the code in part 1 of the assignment. At the top of `page.py` you should have a line that reads:

```
STYLESHEET = 'youface.css'
```

Change it to read:

```
STYLESHEET = '/youface.css'
```

Now go and modify your `loginscreen` function to generate the HTML for a complete login screen. Do this by making an instance of the `LoginPage` class that you created in part 1, then convert it into a string using the `str` function. Return that string instead of the “Hello, world!” test string. When you refresh your browser, you should see a complete, fully rendered login page.

Next, go into `page.py` and customize the page a bit. You may change the title and/or subtitle, and the list of links. Each time you make a change, reload the login page in your browser and you should see your changes.

A form handler

From the login page, if you click on any of the buttons you will get another 404 error page. The browser is trying to submit a form to a handler at the URL `/login`, but you have not created such a handler yet.

Make a new function called `login`, and set it to handle connection requests at the URL `/login`. This will be just like the `loginscreen` handler, but instead of using `@get` use `@post`. When form data is submitted, it uses the POST method to upload the necessary data (we will discuss what this means in class, or you can look it up online).

Create a POST handler for this URL and have it return a simple message (like the “Hello, world!” example from earlier). If you have done it correctly, you should be able to click any of the three buttons on the login screen and see your message displayed back to you in response.

Your code should look something like this:

```
@post('/login')
def login():
    return 'Does it work?'
```

Posted forms submit data that the server can use to implement some action. For our login page, the user’s name and password are submitted, along with the name of the button that the user clicked (“Login”, “Create”, or “Delete”). You can access these values from within your `login` handler using the a Bottle function. For example, to get the name the user typed use:

```
name = request.forms.get('name')
```

The `'name'` passed to the `request.forms.get` function is the name of the `Input` element you created in the `LoginBox` class from part 1. View the source of the login screen, or go back to your `page.py` file to see the names of all of the form values. Note that the submit buttons also have names, and they all happen to have the same name.

Modify your `/login` handler to report the values of all three of these form fields. In other words, your code should get the three values (using `request.forms.get`), and then form a response string that reports what each of the values is, and return that string.

Now you should be able to type in a name and password to the login screen, then when you click “Login”, “Create”, or “Delete” you should get a response that tells you what you typed and which button you clicked.

As always, make sure all these steps are working before proceeding.

Cookies

When a user logs in, you need to have some way to remember the username and password (which you will need each time you contact the database server). You can store these values in a *cookie*. Cookies are sent to the browser by the server, and then the browser sends them back every time a fresh request is issued to the server. You will store the user's name and password in cookies, and then the browser will automatically send them each time the user browses to a new page.

Note that you normally would not store a password in a cookie as it is very insecure, but it will simplify your job. Just make sure you do not use the same password for YouFace as you use anywhere else; you should assume that someone else could find out your YouFace password because of the poor security practices you will employ.

Cookies are a bit like variables. They are stored with a name and a value. To set a new cookie (or change the value of an existing one), use this:

```
response.set_cookie('key', value, path='/')
```

where `'key'` is the name of the cookie, and `value` is the value to be stored.

Use one cookie to store the user's name, and another to store the user's password. Do this every time the `/login` handler is invoked, regardless of which button the user clicked.

To test if this is working, create a new handler (using `@get`, not `@post`) for the URL '/'. It should retrieve the value of your two cookies, which you can do as follows:

```
value = request.COOKIES.get('key', '')
```

This sets `value` to the currently stored value of the cookie with the key `'key'`. The second parameter is a default value; if the cookie does not exist, the function will return an empty string.

Have your handler for `/` report the username and password of the user currently logged in (according to the cookies).

Now go to the login screen, enter a name and password, click one of the buttons, then navigate to '/' and see if it correctly reports your name and password. Note that `/` is the same as:

- http://localhost:8080/

Try going back to the login screen, setting a new name and password, and then checking `/` to see if it was updated successfully.

Now make a logout handler. It should be a `@post` handler, but for now use `@get`. The URL should be `/logout`, and the function should set both cookies to be the empty string.

Navigate to that URL, then go back and reload `/` and confirm that the cookies have been erased.

After erasing the cookies, have the `/logout` handler automatically redirect the browser to `/`. To do this, add this line after setting the cookies:

```
redirect('/')
```

Now try the whole process again. Log in from the `/loginscreen` handler, navigate to `/` to see that your name and password were stored, then navigate to `/logout`. It should automatically redirect you back to `/` where you can see that your name and password have been cleared.

Add the same redirect to the `/login` handler. It should retrieve all the form values, set the two cookies, then redirect to `/`. Of course, it also needs to actually create a user account or delete a user account if that was the requested action, but we will come back to that. For now, just pretend all requests and to log in, and all such requests are successful.

Starting the feed page

The handler for `/` will normally serve the main feed page. Set it to start doing this:

- Start by retrieving the user's name and password from the cookies.
- If the name is not set (it comes back as the empty string), redirect the browser to `/loginscreen`.
- If it is set, create an instance of the `FeedPage` class you created in part 1, convert it into a string, and return that.

- For now, use an empty list for the list of updates and the list of friends, but use the name retrieved from the cookie as the name parameter for `FeedPage`.

Change the `/logout` handler to use `@post` instead of `@get`. Now you should be able to do the following:

- Log in from the `/loginscreen` URL. Clicking any of the three buttons achieves the same result for now.
- When you log in, you are automatically redirected to the feed page with your name displayed.
- When you click the `Logout` button on the sidebar, you are logged out (using the `/logout` form handler), and automatically redirected back to the `/loginscreen` page.

Starting the friend page

The only screen missing is the friend page. This page is a little different than the others. It uses a URL of the form `/friend/Name` where `Name` is the name of the friend. Instead of the friend's name being submitted as part of a form, it is placed directly in the URL. To handle this, you need to warn Bottle that part of the URL is actually a parameter. Your `@get` line should look like this:

```
@get('/friend/:fname')
```

When Bottle calls your handler function, it will supply a single parameter (the string it got from the URL), which should also be named `fname`:

```
def friend(fname):
```

This handler should be very similar to the feed page handler. Have it retrieve the user's name and password, and if they are blank, redirect to the `/loginscreen` page. Otherwise, create an instance of the `FriendPage` class, convert it into a string and serve that to the user. Use the friend's name (not the logged in user's name), and empty lists for the status updates and friends.

You should be able to navigate to one of these pages by typing a suitable URL, e.g.:

- http://localhost:8080/friend/Alice

Following this link should give you a friend page if you are logged in, and redirect you to the `/loginscreen` page if you are not logged in.

Now go back to your feed page handler (`/`) and create a static list of friends when you generate the page. When you load this page, you should see the list of friends in the sidebar, and you should be able to click on each one to see a custom friend page for that friend.

The database server

You can now click around and see all of the screens, but they are not connected to the database. The starter code in `main` connects you to the database, and gives you access to it through the global variable `DB_SERVER`. This is an object with the following methods:

- `newUser(name, password)`: add a new user to the system with the given name and password.
- `deleteUser(name, password)`: delete the given user including the list of friends and all past status updates.
- `addFriend(name, password, friend)`: add someone to the list of friends for the given user. `name` and `password` refer to the user, `friend` is the name of the new friend.
- `unFriend(name, password, friend)`: remove the listed friend from the list of the given user.
- `listFriends(name, password)`: get a list of friends for the given user.
- `setStatus(name, password, message)`: set a new status line for the given user.
- `listStatusFriends(name, password, count)`: return a list of the most recent status updates for the given user and his/her friends. The `count` parameter is a number; at most that many lines will be returned.
- `listStatusUser(name, password, friend, count)`: return a list of most recent status updates for a specific friend of the

given user. Count limits the size of the returned list.

Each of these functions always returns a list. The first element of the list is the status: it is `'success'` if the request succeeded, or `'failure'` if the request failed. If the request failed, the second element in the list is a message about what went wrong. If it succeeded, the second element in the list is a message about it (if there is no other return value for the particular function), or the second element is the promised return value (for all the methods starting with “list”).

Start by trying some of these methods out by hand. Load a python shell, and type these commands to get a database server object:

```
import xmlrpclib
DB_ADDRESS = 'http://youface.cs.dixie.edu/'
DB_SERVER = xmlrpclib.ServerProxy(DB_ADDRESS, allow_none=True)
```

Now try calling the functions listed above as methods of `DB_SERVER`:

```
DB_SERVER.newUser('My Name', 'mypassword')
```

Try each of them. Create and delete user accounts, add friends, set statuses, list statuses, unfriend, etc. The next step will be to call these functions automatically in your web server.

Putting everything together

Now it is time to start connecting URL handlers to the database. Your file already has a database server object called `DB_SERVER` (it was part of the starter code).

Login

Note that the `/loginscreen` handler does not need to connect to the database at all, so it is already fully functional. The first one that needs to be updated is the `/login` handler. What you have already written is mostly correct. The following changes are needed right before the browser is redirected:

- If the user clicked the `Create` button, you should call the `newUser` method to add that user to the database. The name and password are the values should already be retrieving from the login form. E.g.:

  ```
  (status, message) = DB_SERVER.newUser(name, password)
  ```

- If the user clicked the `Delete` button, you should call the `deleteUser` method to remove that user from the database. The name and password are the values should already be retrieving from the login form.

Note that you should still set both cookies regardless of which button was clicked. You should always redirect to `/`, regardless of whether or not the database requests were successful.

Feed page

The `/` feed page handler is next. It should get the name and password from the cookies as before. However, it should not check if they are empty and redirect.

Instead, the handler should just assume the name and password are correct, and call the `listFriends` and `listStatusFriends` database methods (a limit of 25 results for `listStatusFriends` is reasonable) and capture the return values for each. Each of these methods validates the name and password, and will return a failure message if the user account was invalid.

If either method fails for any reason, redirect the user to `/loginscreen`. If they both succeed, use the values the lists they returned to form the `FeedPage` object.

The feed page should render correctly now.

You should also be able to test the login page more thoroughly now. Create a user, and you should go to the feed page. Login as a user, and you should also go to the feed page. Login incorrectly, and you should be redirected back to the login page. If you delete a user, you should also be redirected back to the login page, and subsequent attempts to log in as that user should lead you back to the login page.

Friend page

The friend page handler is next. It is very similar to the feed page handler. The only differences are:

- Call `listStatusUser` instead of `listStatusFriends`, supplying the appropriate set of arguments.
- Generating a `FriendPage` instance instead of a `FeedPage` instance.

For both handlers, the name and password come from the cookies, and the results from the database lead the user to the login page redirect for a failure, or a rendered page for success.

Note that if the friend page renders without the stylesheet information, you probably forgot to change the stylesheet link to `/youface.css` instead of `youface.css`.

Status update

The main screens are all rendered with live updates now, but up to this point there is no way to set status updates for a user (other than talking to the database directly).

Add a `@post` handler for the `/status` URL. It should get the name and password from cookies, get the `status` form variable from the request, and call the `setStatus` method on the database. When finished, it should redirect the user to `/`.

Test this by typing in a new status on the feed page. You should immediately see it updated when you click the `Change` button. Also, the updated status should appear on the feeds of your friends. Finally, when a friend views your friend page, the status should appear there as well. Test all of these cases before proceeding.

Add friend

Up to this point, you have only had the ability to add friends by talking to the database directly. It is not time to make the form on the friend gadget work.

Add a `@post` handler for the `/addfriend` URL. It should get the name and password from cookies, get the name of the new friend from the `name` form variable from the request, and call the `addFriend` method on the database. When finished, it should redirect the user to `/`.

Try adding friends to test this.

Unfriend

The final missing action is the ability to remove friends from your list. This will make the Unfriend box fully functional.

Add a `@post` handler for the `/unfriend` URL. It should get the name and password from cookies, get the name of the new friend from the `name` form variable from the request, and call the `unFriend` method on the database. When finished, it should redirect the user to `/`.

Part II Pass-off

All functionality of the site should be fully implemented. Show the labbie that you can create users, log in, delete your account, and then not be able to log in anymore.

When logged in, you should be able to view status updates, change your status (and see the new status line), and see the status updates of friends. You should be able to add friends, see them in your list, and load a friend page for each one. The friend pages should show the status updates for that friend only, and should allow you to successfully unfriend someone. When you unfriend, all status lines for that user should disappear from your status feed.