Project: Snapgram

In this project you will develop a photo sharing application which we will call *Snapgram*. The basic idea is that each user has a stream of photos with the most recent photo at the front. Users can "follow" other user's streams. A user's feed is the photos (most recent first) from all of his or her followed streams (plus their own photos).

Implement your system using Node.js and Express (including the jade template engine). There are a few more details about the tech stack at the end of this document.

Tip: deploy early and often. See D2L for instructions.

Definitions

A *photo stream* is a chronologically sorted list of all photos uploaded by a user (newest photo first).

A user can *follow* any number of other users's photo stream.

A user's *feed* is a chronologically sorted list of photos from followed streams as well as the user's own stream. Only photos added to a stream since the time the user began following the stream are included in the following user's feed.

If a user unfollows a stream than photos from that were part of their feed from the time that they were following the stream, remain in the feed.

High priority functional requirements

These specifications don't explicitly describe how the various web pages are to look, nor do they describe how the design of the navigation aspects of the page. However, some of the marks for the first deliverable of the project be awarded for a simple, but effective visual design.

Though not explicitly listed as a requirement here, your server will need to serve various **static files** in support of the requirements listed below. For example, any CSS files referenced in your HTML should be served by the server.

0. Unknown path

When a request contains a method and path that is not one of the method paths described in this document, the server responds with a status code of 404 (Not Found).

Request

```
GET /unknown/path/to/file.html
```

Response

Code: 404 Not Found
Body: HTML document with suitable message

Similarly, when receiving request for a photo (see feature 9) or a user stream (see feature 7) that does not exist, the server responds with a status code of 404.

Request

GET /users/123

Response (assuming there is no user 123)

Code: 404 Not Found
Body: HTML document with suitable message

1. Server error

When an error is encountered (e.g., a database connection error) that prevents an otherwise valid request from being processed, the server responds with a status code of 500 (Internal Server Error).

Response

Code: 500 Internal Server Error
Body: HTML with suitable message

2. User registration

Users register (or sign up) with the site by providing their full name, a username and a password. The system encrypts the password before it is stored in the database.

Request

GET /users/new

Response

Code: 200 OK
Body: HTML for the sign up form

The form has an inputs for a user's full name, desired user name, a password and a submit button (with value Sign up). When the form is submitted the following request is sent to the server.

Request

POST /users/create
Body: name value pairs from form

The server verifies that a password and a (valid and unique) user name were supplied. If so, the user is created and saved in db. The new user is also

logged in by creating a new session and setting a cookie with the generated session id (see feature 3 for more details). The response redirects the user agent to the feed of the new user.

Response

```
Code: 302 Found
Headers:
   location: /feed
   set-cookie: sid=<session id>
```

If the user can not be created (because the above verification step fails) the HTML sign up form is displayed again with an appropriate error message displayed at the top.

Response

```
Code: 302 Found
Headers:
   location: /users/new
```

Note that each user is identified by a numeric id (likely generated by the database) that is used internally but also in a few URLs.

3. User login

Registered users can log in using an user name and password. The system sets a cookie in the user's browser and stores that cookie on the server to identify the user's session. A request is known to be from a logged in user by the presence of a valid cookie (i.e., a cookie that matches one stored on the server).

Request

```
GET  /sessions/new
```

Response

```
Code: 200 OK
Body: HTML for the login form
```

When the user enters their user name and password, the following request is sent

```
POST /sessions/create
Body: form values including user name and password
```

If the user name and password are valid, a new session is created and the session id is sent to the user agent using the set-cookie header.

Response

```
Code: 302 Found
Headers:
   location: /feed
```

set-cookie: sid=<session id>

If the user name and password do not match a user in the database, the login form is displayed again with an appropriate error message displayed at the top.

Response

Code: 302 Found
Headers:
location: /sessions/new

4. Redirect to login form

As mentioned above, a request is identified as coming from a logged in user by the presence of a valid session id (sid) in a cookie sent with the request. With a few exceptions, requests from a user that is not logged in are redirected to the login page (see feature 3).

Request (example)

GET /feed

Response (if user is not logged in)

Code: 302 Found
Headers:
location: /sessions/new

The exceptions to this rule are requests associated with signing up and logging in which obviously don't require a logged in user.

5. Feed

After a successfully logging in a user is redirected to their feed as defined above. For a logged in user, requests for the root path / are also redirected to the feed.

Request

GET /

Response

Code: 302 Found
Headers:
location: /feed
set-cookie: sid=<session id>

Request

GET /feed

Response

Code: 200 OK
Body: HTML page displaying the user's feed

This HTML page shows thumbnails of the most recent **30** photos in the user's feed. The caption under each thumbnail shows the full name of the user who uploaded the photo, along with a short description of how long ago the photo was uploaded (e.g., 2 mins ago or 3 weeks ago). In the caption, the full name of the user is a link to that user's stream (see feature 7).

At the bottom of the user's feed (assuming there are more photos to show) is a more link. This link sends the following request.

Request

GET /feed?page=2

Response

Code: 200 OK
Body: HTML page displaying the next 30 photos

The more link is included on each page for as long as there are more photos to show.

6. Uploading Photos

Users can upload a photo to their personal photo stream. The sequence of requests is described here.

Request

GET /photos/new

Response

Code: 200 OK
Body: HTML page displaying the upload form

When the form is submitted by the user, the following request is sent to the server.

Request

POST /photos/create
Body: multipart form data that includes the photo

When the user submits the form, the server verifies that a file was uploaded and that that file is an image. If so, the photo is saved in the user's stream along with a time stamp indicating when the photo was uploaded and a numeric identifier. The image file itself can be saved to the file system using the ID rather than the original filename (to avoid name collisions).

Response

Code: 302 Found

Headers:
location: /feed

If the submission is not an image (or there is not file at all) the system responds with a redirect to the upload form along with a suitable message.

Response

Code: 302 Found
Headers:
location: /photos/new

7. User Stream

In the following request :id refers to the (numeric) id for a user.

Request

GET /users/:id

Response

Code: 200 OK
Body: HTML page displaying the user's stream

This HTML page shows thumbnails of the most recent **30** photos in the user's stream. The caption under each thumbnail shows the full name of the user who uploaded the photo, along with a short description of how long ago the photo was uploaded (e.g., 2 mins ago or 3 weeks ago). In the caption, the full name of the user is a link to that user's stream.

Additional pages of photos in the stream are accessible from more link at the bottom of the page as described above in feature 5.

Request

GET /users/:id?page=2

8. (Un)Follow a Stream

The page for a user's stream (see feature 7) should have a follow link at the top. Or if the current user is already a follower of that stream, there should be an unfollow link at the top.

Request

GET /users/:id/follow

Response

Code: 302 Found
Headers:
lcoation: /users/:id

Request

```
GET /users/:id/unfollow
```

Response

```
Code: 302 Found
Headers:
   lcoation: /users/:id
```

The semantics of following and unfollowing are described above.

9. Serving Images

The pages for a user's feed or stream include thumbnails of photos, which are 400px wide (regardless of the size of the original). hotos with width smaller than 400px need to be upscaled to 400px. Resizing must retain the aspect ratio of the photos.

Thumbnails are served in response to requests that identify the numeric database id for the photo and the extension indicating the type.

Request

```
GET /photos/thumbnail/:id.:ext
```

For example if the a png photo has the numeric id 123 the request would look like.

Request

```
GET /photos/thumbnail/123.png
```

Response

```
Code: 200 OK
Headers:
   content-type: ...
Body: 400px wide image
```

A request for the original photo (rather than the thumbnail) looks similar.

Request

```
GET /photos/:id.:ext
```

Bulk upload requirements

These requirements are important for the TAs to be able to test your system using a script. These features may also facilitate your own testing. For some modest amount of security, the TAs will give each team a password that needs to be included with each bulk request.

The first step in the testing sequence will be to reset (or clear) your server's database.

Request

GET /bulk/clear?password=:password

The server then erases all of the data in the database (though not the schema) and responds as follows.

Response

Code: 200 OK
Body: plain text message (DB cleared)

Next the script will upload bulk user data. You can safely load the user data into memory (for convenience in parsing it) because if it is too large the script will send multiple requests.

Request

POST /bulk/users?password=:password
Body: JSON user data formated as below

Format

```
[{id:1, name:'jill', follows:[3,4,5], password:'abcdef'},
 {id:2, name:'bill', follows:[2,4,5,11], password:'abcdef'},
 ...]
```

Next the script will upload bulk user stream data (i.e., photos). The node.cs server we are using for the course will store a corpus of photos in a publicly accessible location. The bulk upload of photos will not include actual photos just the path name of the shared photos. Again, you can safely load the photo data into memory (for convenience in parsing it) because if it is too large the script will send multiple requests.

Request

POST /bulk/streams?password=:password
Body: JSON photo data formated as below

Format

```
[{id:1, user_id:2, path:'/shared/1.png', timestamp=1392405505782},
 {id:2, user_id:5, path:'/shared/2.png', timestamp=1392405510031},
 ...]
```

Treat the timestamp (given as a number of milliseconds since epoch) as the upload time and date for the photo.

Once the user and photo data is loaded, the script will test the performance of the system by sending various requests as described in the functional requirements section.

Additional (low priority) functional

requirements

Do not add these features unless you are confident that the above requirements are complete. Only a small number of marks will be assigned to these.

Share. Below each photo will be a share link (unless the current user has already shared it) that allows a user to add a photo to their stream even if they are not the one who has uploaded the photo. In this case any followers of the stream will see that in their feeds based on the share time rather than the upload time. Under each shared photo will be the name of the user who uploaded it as well as the name of the user who shared it:

Jill Smith (shared by Bill Johnson)

Both names should be links to the respective user streams. Note that a photo should appear at most once in any feed, regardless of how many times it has been shared. Also, the request to share a photo should be sent asynchronously.

Other extra requirements may be added later (e.g., a comment feature, or a like feature).

Stack

For this project you are to use Node.js and the Express framework. You are also to use the provided mysql database. Below is a list of additional packages you may use. If would like to use other modules, you must first get permission from one of the TAs via the D2L discussion group.

- You may use an Express plugin for logins as long as you can configure it to work in the way described in feature 3 above.
- You may use an ORM (such as node-orm), as long as you use the mysql database provided. The basic node-mysql package is already installed and available for you to use.
- GraphicsMagic (already installed) and the node gm package.
- If you choose, you may use the very popular Bootstrap front-end framework.

Note that node.cs.ucalgary.ca where you are to deploy your systems is not backed up. So use it only for deployment and testing not development.