EECS 485 Project 2:   
Authentication and Sessions

Due 9pm Thursday, Oct 13, 2016

In this project, you will continue working on the photo album website developed in project 1. However, do not touch the files in the p1 sub-directory. Make another sub-directory called p2, and copy the files from p1 into the p2 sub-directory and work on the files there. Be sure to remove the .git folder and connect to the new repo given for project 2. Modifying your p1 folder after the project deadline is in violation of the honor code. It can also hurt the grading process. By the end of this programming assignment you will learn how to authenticate users and maintain sessions.

Finally, make sure to change your routes to '{secret-key}/*p2*/'. **In general, we recommend using the blueprint argument** [**url\_prefix**](http://flask.pocoo.org/docs/0.10/blueprints/#registering-blueprints) **for specifying these routes.**

[Part 1: Getting started](#_dy428kri557i)

[Part 2: Build the website](#_1ggm0aq284iy)

[Part 3: Validation](#_2z9epg6nc4b)

[Grading](#_np3se47ex91u)

# Getting started

This project is about personalization. The first step to doing any kind of personalization is to keep track of who is browsing your site. In class we discussed how HTTP is a stateless protocol, which cannot itself retain data from one request to the next. The way to maintain state from one page request to another is by using **sessions**. In this project we will add a login page to the site. Users will only need to type their username and password once. Thereafter, your website will use session variables to determine who the logged in user is.

Pages that are *sensitive* require users to login before they can view those pages. The rest of the pages will remain public and will not require a username or password to be viewed. Whenever a user tries to enter a sensitive page you should make sure that he/she has the privileges to view it. This is done by checking if the user has a valid session, and if so, whether the user is authorized to view the page.

Some pages do not require user authentication or sessions (e.g., a public home page or a create account page). Some other pages only require that the user be authenticated and are not dependent on who the user is (e.g., a logged in home page). Others may provide access depending on who the user is and whether he/she is permitted to access that page (e.g., someone's private album).

**In short, when a user requests a URL, you should:**

* If the page is public, view the page
* If the page is not public (it is sensitive):
  + If there is no session, redirect to the login page
  + If there is a session
    - If the user has permission for the page, view the page
    - If the user doesn’t have permission, return a [403 error](http://flask.pocoo.org/docs/0.10/patterns/errorpages/).

These sites contain useful tutorial and reference information for what you'll be implementing in this project.

* [HTTP Cookies](http://en.wikipedia.org/wiki/HTTP_cookie)
* [Sessions: Python](http://flask.pocoo.org/docs/quickstart/#sessions)

*Tip:* I[n](http://flask.pocoo.org/docs/quickstart/#sessions) [a](http://flask.pocoo.org/docs/quickstart/#sessions)ddi[t](http://flask.pocoo.org/docs/quickstart/#sessions)i[o](http://flask.pocoo.org/docs/quickstart/#sessions)n[,](http://flask.pocoo.org/docs/quickstart/#sessions) [y](http://flask.pocoo.org/docs/quickstart/#sessions)ou [s](http://flask.pocoo.org/docs/quickstart/#sessions)h[o](http://flask.pocoo.org/docs/quickstart/#sessions)u[l](http://flask.pocoo.org/docs/quickstart/#sessions)da[v](http://flask.pocoo.org/docs/quickstart/#sessions)oidh[a](http://flask.pocoo.org/docs/quickstart/#sessions)ndl[i](http://flask.pocoo.org/docs/quickstart/#sessions)n[g](http://flask.pocoo.org/docs/quickstart/#sessions) [r](http://flask.pocoo.org/docs/quickstart/#sessions)aw [U](http://flask.pocoo.org/docs/quickstart/#sessions)RLsin [F](http://flask.pocoo.org/docs/quickstart/#sessions)las[k](http://flask.pocoo.org/docs/quickstart/#sessions).I[n](http://flask.pocoo.org/docs/quickstart/#sessions)s[t](http://flask.pocoo.org/docs/quickstart/#sessions)ead[,](http://flask.pocoo.org/docs/quickstart/#sessions) [u](http://flask.pocoo.org/docs/quickstart/#sessions)s[e](http://flask.pocoo.org/docs/quickstart/#sessions) [url\_for](http://flask.pocoo.org/docs/0.11/api/#flask.url_for), [redirect](http://flask.pocoo.org/docs/0.11/api/#flask.redirect), [abort](http://flask.pocoo.org/docs/0.11/api/#flask.abort),ando[t](http://flask.pocoo.org/docs/quickstart/#sessions)h[e](http://flask.pocoo.org/docs/quickstart/#sessions)rhel[p](http://flask.pocoo.org/docs/quickstart/#sessions)e[r](http://flask.pocoo.org/docs/quickstart/#sessions) [m](http://flask.pocoo.org/docs/quickstart/#sessions)eth[o](http://flask.pocoo.org/docs/quickstart/#sessions)d[s](http://flask.pocoo.org/docs/quickstart/#sessions) [t](http://flask.pocoo.org/docs/quickstart/#sessions)od[i](http://flask.pocoo.org/docs/quickstart/#sessions)s[p](http://flask.pocoo.org/docs/quickstart/#sessions)layanddir[e](http://flask.pocoo.org/docs/quickstart/#sessions)ct [u](http://flask.pocoo.org/docs/quickstart/#sessions)ser[s](http://flask.pocoo.org/docs/quickstart/#sessions).

# Part 1: Setting up the database

In this project, we will be adding the ability to set access privileges for albums. In order to support this, we will need to modify our database accordingly.

## Rules for Access

* Public albums are accessible to both logged in users and unauthenticated visitors.
* Private albums are accessible only to those users that have explicit access to that album (and the owner of the album). A user will have access to another user's private album if and only if there exists a tuple (albumid, username) in the AlbumAccess table (see below).

For example, User Claire creates albums Alpha (public), Bravo (private), and Charlie (private). Claire grants User Prateek access to Bravo. Prateek creates album Delta (private).

Prateek can view albums Alpha, Bravo, and Delta on his logged-in index page. These are the albums Prateek has permission to see.

Prateek can view **only** album Delta on his logged-in /albums page, by clicking on the link to 'My Albums'. These are the albums Prateek has permission to **edit.**

Claire has permission to view the albums Alpha, Bravo, and Charlie, both on her logged-in index page, and /albums page. She can edit these albums. However, she cannot view Delta.

## Database Changes

Just like Project 1, you need a sql folder with the files tbl\_create.sql and load\_data.sql. You can/should use your Project 1 files as a starting point. The changes below should be made in these sql files.

### Album Table

You need to add an access attribute to this table, so the new scheme for Album will be

* Album ( **albumid**, title, created, lastupdated, username, access )

access specifies whether access to the album should be limited to a set of users indicated in the AlbumAccess table (described below). It only takes values of public or private. It should be an enum. More information about MySQL enums can be found [here](https://dev.mysql.com/doc/refman/5.5/en/enum.html).

The following albums should exist in the following order (same as p1) and access values:

* "I love sports" (public)
* "I love football" (private)
* "Around The World" (public)
* "Cool Space Shots" (private)

Please make sure your albums have the correct ordered id: these should be the same as p1 and consistent with the order shown above. Your website may contain other users and albums, but please ensure that the above users and albums exist.

### AlbumAccess Table

* AlbumAccess (**albumid**, **username**)

This relation indicates the users who have access to each specific album. The owner of an album should **not** appear in AlbumAccess for that album.

When an album is deleted, all permissions given for that album should be deleted.

### Changing the Password Field

In addition, we will need to have a larger size for the password (varchar(256)) field in the User table.

In this project, we will store a user’s password like this: <algorithm>$<salt>$<hash>, where <algorithm> is the name of the hash algorithm (“sha512”), <salt> is a 64-bit hex [uuid](https://docs.python.org/2/library/uuid.html), and <hash> is the encrypted password. ‘$’ is a literal dollar symbol. [Here](https://crackstation.net/hashing-security.htm) is an explanation on why we salt and hash passwords.

Here is an example of password encryption.

import hashlib

import uuid

algorithm = 'sha512' # name of the algorithm to use for encryption

password = 'bob1pass' # unencrypted password

salt = uuid.uuid4().hex # salt as a hex string for storage in db

m = hashlib.new(algorithm)

m.update(salt + password)

password\_hash = m.hexdigest()

print "$".join([algorithm,salt,password\_hash])

sha512$523bbfca143d4676b5ecfc8ee42aca6d$fae41640d635cb42c3631e5a66a997e6f6ebfd25f6bb3f9777107d848c24bd2db9767242e803a881dbc5af73ddbf7ee80d1d855db2568061bfb2ca21fcf2dd5f

When initializing your database, you should store the passwords (original passwords are the same as p1) in this format (note that the actual password will be different because you will be generating a unique salt):

username = sportslover

password = paulpass93

database = sha512$8ec61415f1eb4afba45fa95e164a73e5$a8156f5e122a936e55512ccad145e72581c20853d8ceee8fc4ab535bead173dfb6625dd1d0eaccc9ace73008c135ef5eecb0b452470d007fde088602659ad9a2

username = traveler

password = rebeccapass15

database = sha512$1c662feb81e84cd78cf8d6a96e912ebb$eed150f49e6669c4aee79b0f1ed238ec557e8a6dc1af8c8b4dd393a1a6f0926b97bb537fc7a7af95db36982eaa90a313d4968cdc03112321e9dbb3c4aba65337

username = spacejunkie

password = bob1pass

database = sha512$523bbfca143d4676b5ecfc8ee42aca6d$fae41640d635cb42c3631e5a66a997e6f6ebfd25f6bb3f9777107d848c24bd2db9767242e803a881dbc5af73ddbf7ee80d1d855db2568061bfb2ca21fcf2dd5f

# Part 2: Build the website

## Session Management

Sessions allow us to reliably keep track of User information while the User

is browsing our website. **This project is centered around the use of sessions.**

A good, concise explanation on sessions: <http://stackoverflow.com/questions/3804209/what-are-sessions-how-do-they-work>

You will want to maintain one session variable: username. The username stores the username of the authenticated user.

### Sessions in Python:

In Flask, sessions are started automatically. The session variables can be accessed as such:

session['username']

In order for sessions to work properly, however, a secret key must be set. Please refer to the [Flask docs](http://flask.pocoo.org/docs/0.10/quickstart/#sessions) for more information about how to use sessions (you will want to put the key in your app.py file). Be sure that sessions are imported when attempting to use them. Sessions work similarly in other Python frameworks.

The User's session is only maintained and used on the server-side; There is no need to write client-side code to maintain sessions. For simplicity in this project, assume cookies are **always enabled**.

## URL Endpoints

What follows is a list of the url endpoints that you should create in your application. You should have created some of these for p1. A **page is either in a public state (no session involved) or sensitive state (need session to manage page).**

As you work on these, check “Grading and Deliverables” below to ensure you have all the appropriate HTML elements and ids.

#### Default home page: / [public]

This page contains a welcoming message and information about the website. It should have links for new users to join as members (/user), or sign-in (/login) if they already are members. There should also be links from this homepage to all the *public* albums of all users. There should also be links (as in Project 1) from this page to a page with a page of public albums for each user (/albums?username=<username>).

Logged in homepage: / [sensitive]

This page is the home page for a user who has already logged in. Make sure for all pages in a logged-in state, you clearly display the message "Logged in as <firstname> <lastname>" (in the header). This page and all subsequent logged-in pages should have a navigational interface with links to Home (this page), Edit Account (/user/edit), My Albums (/albums) and Logout (/logout); this might be good to put in your base.html. The main body of the page should have a list of all the accessible albums. Accessible albums include public albums, your own private albums, and private albums which you have been granted access to by the owner.

#### New User page: /user [public]

This page is for user creation. It contains a form that should POST to /user with the following information for account creation:

username: “<username>”,

firstname: “<firstname>”,

lastname: “<lastname>”,

password1: “<password1>”,

password2: “<password2>”,

email: “<email>”

Make sure the password field does [not display uncensored text](http://resources.bravenet.com/tutorials/forms/passwordinput.php) and that there are two password fields for verification. Follow the validation rules set forth in Part 3 that describe the set of permissible data. You should use server-side validation for all data validation.

If a User provides all valid information, create the User by inserting their information into your database. Make sure that the password is hashed and salted as described in Part 1 before it is inserted.

If a session already exists, redirect the user to /user/edit. Otherwise, after adding the user to the database, redirect to /login and allow the new user to log in with their new credentials (do not log them in automatically after signup).

#### Edit Account page: /user/edit [sensitive]

The user should be able to change his/her firstname, lastname, password and email address (but not username). You should only change one field per request. This likely means your implementation will have forms for each field, all POSTing to the same route.

An example POST to /user/edit is:

“password1”: “newpassword1”,

“password2”: “newpassword1”

OR

“firstname”: “Mark”

Again, validate the input values on the server (details in Part 3).

#### User Login page: /login [public]

Here, a non-authenticated user can enter their username and password to login to a specific user account and become authenticated. You should use a form with username and password fields to send authentication information to the server for validation. A POST request to /login should have the following body to sign in the user:

username: “<username>”,

password: “<password>”

Refer to Part 3 "Validation" for more details on how to notify users when they incorrectly attempt a login. If one of these login issues occurs, the /login page should be returned again, but this time with a descriptive error message at the top.

If login is successful, the user should be redirected to the main logged-in page: /.

#### My Albums page: /albums [sensitive]

This page is similar to its corresponding route in Project 1, in that it contains links to albums which the current User owns, as well as a link to the /albums/edit route. Note that, instead of using a URL parameter to input the username of the User’s albums that we want to see, we are instead using the current session. You can navigate here by clicking on the ‘My Albums’ button at the index page.

#### Public Albums of User:/albums?username=<username> [public]

This page shows all of the public albums of the user specified by the username GET parameter, whether or not you are logged in. Here, you find links to view the albums, but no links to edit them. You can navigate here from the index page.

#### Albums page: /albums/edit [sensitive]

This is the /albums/edit page from Project 1. This page allows the user to add new albums, view existing albums, delete them or edit them. Remember that deleting an album should also involve deleting pictures in the album.

A few notes:

* A user can only edit albums they created.
* There is no username parameter because we get that from the session cookies.
* All new albums have private access by default.
* Deleting an album deletes user permissions for the album.

The interface for this page should look as in Project 1, except that it may be helpful for you to add an indicator for whether an album is public or private.

#### Edit Album page: /album/edit?albumid=<albumid> [sensitive]

￼￼￼At the top of this page the user should be able to change the album access permissions. If the album is private, there should be some way the user can edit a list of **other users** to whom he/she would like to give explicit access to view this album (*Note*: the permissions of the album’s owner cannot be altered). If an Album becomes public, **all users that don't own the album but have permission to view it should lose said permissions.**

You should also list the pictures in the album. Users should be able to delete pictures from the album as well as add new pictures.

Users should also be able to click on individual images and be directed to /pic from Project 1. Make sure to keep the Album.lastupdated field in the database updated whenever Album.access is changed (as well as all album updates from Project 1).

When granting a user permission, the form data should have the following:

op: “grant”,

albumid: <albumid>,

username: “<username>”

When revoking a user’s permission, the form data should have the following:

op: “revoke”,

albumid: <albumid>,

username: “<username>”

When modifying an album’s public/private access, the form data should have the following:

op: “access”,

access: “public” OR “private”,

albumid: <albumid>

The interface for the new portions of /album/edit should appear roughly as below:

<form>

<input type=”radio” name=”access” value=”public” />Public<br />

<input type=”radio” name=”access” value=”private” />Private<br />

<input type=”submit” value=”Submit” />

</form>

<table>

<tr><th>Username</th><th>Update Access</th></tr>

<tr><td>sportslover</td><td>[Revoke]</td>

<tr><td>traveler</td><td>[Revoke]</td></tr>

<tr><td>New: \_\_\_\_\_\_\_\_\_\_\_\_\_\_</td><td>[Add]</td></tr>

</table>

#### View Album page: /album?albumid=<albumid> [sensitive/public]

This page displays the thumbnail view of an album’s pictures just like the previous assignments. You can view this page if the album is public, you own the album, or the album is private and you have been given explicit access. If the user is logged in and does not have access to the album, a 403 error should be returned. If the user is not logged in and the album is private, they should be redirected to /login.

The album title should be at the top, along with the album's owner. The photos should be displayed in sequence order, each with its date, and a caption (if caption exists for that photo). Similar to project 1, clicking each photo should take you to /pic?picid=<picid>.

#### View picture page: /pic?picid=<picid> [sensitive/public]

This page displays a picture just like the previous assignment. It should have the caption, full-sized picture and links to previous and next picture.

You must be able to edit the caption if it is part of an album that you own. To edit the caption, make a form that POSTs with the following information:

op: “caption”,

picid: “<picid>”,

caption: “<new\_caption>”

If you do, make sure to update the Album.lastupdated field for the album that pic is in.

You can view this page if the picture is part of a public album, you own the album it is a part of, or it is in an album that is private for which you have been given explicit access. If the user does not have access to the album this picture is in, they should not be able to see the picture; an error 403 should be returned. If the user is not logged in and the album is private, they should be redirected to /login.

#### Logout page:/logout [sensitive]

This should destroy the session and redirect to the default home page. It should be a POST request with an empty body (no key/values in the form).

# Part 3: Validation

Your site must enforce the following rules, and the Route(s) column describes in which route you should be checking for each error:

|  |  |  |
| --- | --- | --- |
| **Route(s)** | **Description of Error** | **Exact Text to Return to Users** |
| /user | No field (username, firstname, lastname, password1, and email) is allowed to be left blank (be an empty string) (don’t worry about checking password2) | “<field> may not be left blank” |
| /login | No field (username, password) is allowed to be left blank (be an empty string) | “<field> may not be left blank” |
| /user  /user/edit | Except for password and email, all fields (username, firstname, and lastname) have a max length of 20 | "<field> must be no longer than 20 characters" (If lastname was too long, for example, the error would be “Lastname must be no longer than 20 characters”) |
| /user | The username must be unique (case insensitive) | “This username is taken” |
| /user | The username must be at least three characters long | “Usernames must be at least 3 characters long” |
| /user | The username can only have letters, digits and underscores | “Usernames may only contain letters, digits, and underscores” |
| /user  /user/edit | The password (check on password1, not password2) should be at least 8 characters long | “Passwords must be at least 8 characters long" |
| /user  /user/edit | The password (again, on password1) must contain at least one digit and at least one letter | "Passwords must contain at least one letter and one number" |
| /user  /user/edit | The password (again, on password1) can only have letters, digits and underscores | "Passwords may only contain letters, digits, and underscores" |
| /user  /user/edit | The first and second password inputs must match | “Passwords do not match” |
| /user  /user/edit | Email address should be syntactically valid (see below for more information) | “Email address must be valid” |
| /user  /user/edit | The email has a max length of 40 | “Email must be no longer than 40 characters” |
| /login | Username does not exist | “Username does not exist” |
| /login | Password is incorrect for the specified username | “Password is incorrect for the specified username” |

For validating emails, here is a regular expression which should check email validity well:

import re

if not re.match(r”[^@]+@[^@]+\.[^@]+”, email):

# handle an invalid email address

In the event any of the validation rules are broken, redirect to the page that the information was entered on with the corresponding error messages above.

Make sure that for each error message, you have one paragraph (<p>). If you have no errors, **do not have any error paragraphs.** Appropriate validation errors must be found in your page in the following form:

<p class=”error”>

\*\*Error message goes here\*\*

</p>

Where \*\*Error message goes here\*\* is replaced with the appropriate error message. For example:

<p class=”error”>

Lastname must be no longer than 20 characters

</p>

**You can assume that the user is acting in good faith: your goal is to prevent users from adding bad usernames/passwords, not to guard against motivated attackers who want to sneak a** [**strange entry**](http://en.wikipedia.org/wiki/Code_injection) **into your password database (which means you do not need to check things beyond the above rules).**

# Grading and Deliverables

Make sure that all these element IDs are present in your HTML templates:

* / (not logged in)
  + The link for login should have id home\_login
  + The link for account creation (/user) should have id home\_user\_create
  + The links to public albums should have ids album\_<albumid>\_link
* /user
  + The input for username should have an id new\_username\_input
  + The input for firstname should have an id new\_firstname\_input
  + The input for lastname should have an id new\_lastname\_input
  + The input for email should have an id new\_email\_input
  + The input for your first password field should have an id new\_password1\_input
  + The input for your second password field should have an id new\_password2\_input
  + The submit button for updating should have an id new\_submit
  + Follow the rules for validation input error as specified in the validation section; all error <p>’s need to have class error and the correct error message
* /user/edit
  + The input for firstname should have an id update\_firstname\_input
  + The submit button for firstname should have id update\_firstname\_submit
  + The input for lastname should have an id update\_lastname\_input
  + The submit button for lastname should have id update\_lastname\_submit
  + The input for email should have an id update\_email\_input
  + The submit button for email should have id update\_email\_submit
  + The input for your first password field should have an id update\_password1\_input
  + The input for your second password field should have an id update\_password2\_input
  + The submit button for password should have id update\_password\_submit
  + Follow the rules for validation input error as specified in the validation section; all error <p>’s need to have class error and the correct error message
* /login
  + The input for username should have an id login\_username\_input
  + The input for password should have an id login\_password\_input
  + The submit button for logging in should have id login\_submit
  + Follow the rules for validation input error as specified in the validation section; all error <p>’s need to have class error and the correct error message
* /albums
  + Each link to /album?albumid=<albumid> should have id album\_<albumid>\_link
* /albums/edit
  + Each link to /album/edit?albumid=<albumid> should have id album\_edit\_<albumid>\_link
* /album
  + Each link to /pic?picid=<picid> should have id pic\_<picid>\_link
* /album/edit
  + Each link to /pic?picid=<picid> should have id pic\_<picid>\_link
  + The radio button for public album toggle should have id album\_edit\_public\_radio
  + The radio button for private album toggle should have id album\_edit\_private\_radio
  + The submit button for toggling album access should have id album\_edit\_access\_submit
  + Each “remove access” submit button in the access table should have id album\_edit\_revoke\_<username>
  + The text input for granting new access should have id album\_edit\_grant\_input
  + The submit button for granting new access should have id album\_edit\_grant\_submit
* /pic
  + The <p> that shows the caption should have id pic\_<picid>\_caption
  + The text input for editing the caption should have id pic\_caption\_input
  + The submit button for editing the caption should have id pic\_caption\_submit
* <all signed in pages>
  + The “Nav Home” link should have an id nav\_home¨
  + The “Edit Account” link should have an id nav\_edit
  + The “My Albums” link should have an id nav\_albums
  + The “Logout” submit button should have an id nav\_logout

*Note*: While some of the ids from project 1 were explicitly specified above, you should maintain **all** of the ids we gave you in project 1 (except where noted as different); we can and will test you on them!

## Deploy

You should deploy your code using the same instructions as in Project 1, but remember that your url should be {secret}/p2 instead.

## Code

Submit the following files to the autograder:

* source.tar.gz A tar archive containing your application source code.

How to create a tarball of your source code, from your git repository:

git archive --format tar.gz HEAD > source.tar.gz

We will post a link to the autograder when it is ready for this project.

In the README.md at the root of your repository please provide the following details:

* Group Name (if you have one)
* List the contribution for each team member:   
  User Name (uniqname): "agreed upon" contributions
* Any need-to-know comments about your site design or implementation.

We will check the p2 URLs for your new secure photo album website. **Remember that in this project, we are hashing passwords, so you will have to calculate the data for the initial input**.

As mentioned before, **Remember to commit your code into GitHub and the server, please do not modify your code or touch files in the p2 sub-directory after the due date - either on the repo or the server**, or else we will assume your submission is late.