Password-Based Authentication

Description

Your previous assignment was to build an AppEngine app that stored a few things and returned them from a JSON-enabled endpoint when requested. You also have some mutating functions that allow you to add and (optionally, until now) delete existing events.

For this assignment, you will add authentication so that all of the events are tied to one and only one user. That user must be logged in to see their events. This means you will create a registration form, a login form, and send and receive session cookies.

While it is tempting to use the built-in User class (in Python, at least, other languages have similar facilities), for this assignment you will be rolling your very own authentication. If a person is not logged in, they will be offered a login screen, and if they are not registered, you will provide a way for them to sign up with their own chosen username and password.

Because your existing data is not attached to a user, you will need to create a user for yourself and then migrate the existing data over to it. Migrating data is part of the assignment, so you will need to demonstrate code that does the migration, along with a written strategy for how it is used.

With Google AppEngine's datastore, you will probably want to use the user as a parent entity for the event data. Before, you used a constant parent entity, and now you will set up some kind of user model that becomes the parent of all events.

If you have not successfully implemented all of the necessary facilities in the previous lab, a starting implementation will be provided for you.

As with the previous lab, you will turn in code and demonstrate functionality using a public cloud link to your working system.

Grading Rubric

For this lab you will need to implement basic password authentication and user registration based on some kind of user token (username, email, whatever). You will need to at least have login working with an opaque session token used for ongoing authentication to get a B-.

Minimum Requirements

- HTML/JavaScript still served statically, dynamic requests through JSON requests.
- All interactions are over HTTPS.
- At least one user can log in with a password.
- When not logged in, the user is automatically directed to a login page.
- Logging in produces a secure session token that is stored as a cookie, and that cookie allows users to continue using the site without interrupting them with a login form.
- All JSON endpoints are protected, so they cannot be accessed without a session token.
- Only secure derivatives of passwords are stored in the database. No plaintext.
- Original data (that was not previously attached to a user) is migrated to a user.

My suggestion for storing password derivatives (for Python users) is to include Python bcrypt libraries inside your project directory, maybe under a "lib" folder. See this StackOverflow answer for one way to do that.

Additional Points

Do 4 out of 5 of these to get 100% after the minimum requirements are met.

- Provide a registration page that allows the user to provide a username and password, then immediately logs them in and redirects them to the main page.
- Provide a short design document indicating how you migrated your old data to one of the new users so that it shows up for them when they log in.
- Add a logout button that invalidates the session token, removes the cookie, and redirects the user to the login page.

- Set your JSON calls up to redirect to the login page when they fail due to a missing (or expired) session token.
- Make your session tokens expire after 1 hour. Enforce this by deleting the cookie when an expired session token is used. One way to go about this is detailed in this StackOverflow answer.

Non-Storage of Passwords

As part of this assignment, you will be creating a form that sends a username and password from a user's browser to your service. This can happen when registering a new user or merely logging in an existing one. Because these will be sent over the Internet, you *must* ensure that the connection when POSTing this information to the server is always secure.

To do this in AppEngine, you can set "secure: always" in each handler. That will force SSL. Note that it won't necessarily **redirect** you from HTTP to HTTPS, so you might need to ensure that you're typing "https://" in your URL bar as you test.

If you wanted to be especially careful, you could compute an appropriate hash of the password using a cryptographic hash like SHA-256 *in the browser* and send that hash to the server. Note that this does not mean you can avoid using SSL: if someone hacks your site or is otherwise able to inject their own JavaScript into the transmission (because it's an unsecured channel and they could feasibly execute a man-in-the-middle attack), then they would be able to exfiltrate the password with some malicious scripting while still preserving overall site functionality. In short, you always want end-to-end encryption when sending sensitive data, even if it's a computation that **comes from** sensitive data, to a server.

Note that it is not at all typical to compute hashes in the browser like this. For one thing, you can never guarantee that a company is doing that, and for another, SSL means you already trust the endpoint and have an encrypted channel to it. Therefore, passwords are just sent in plaintext over the encrypted channel. What's special is what happens when they *arrive*: they are not stored that way.

Your server should not store the actual password, but a cryptographic derivative (e.g., a secure hash) of it. When testing whether a username and password are correct, therefore, you will not compare passwords, but two hashes: one stored, and one computed from what the user typed during login. Use key stretching, something like Bcrypt or Argon2. Libraries exist for these: don't make your own.

User Records

With AppEngine, it is possible to create what's called a "parent model". We used a single universal one in the previous lab to ensure ordered database operations (everything was in a

single entity group). For this assignment, you will create a model whose key is a user ID. The user model will contain a hash of the user's password, and potentially other things that you think it ought to contain.

Shared Session Tokens

If you require a user to login every time they do everything on the site, you and your users are going to have a bad time. Therefore, once you have identified the user as being in your system, and the password is correct, you should create a random secret token that you then pass back in the Set-Cookie header of the response. Take very careful note of the domain of the cookie. Since you are likely going to be operating on somedomain.appspot.com, and the last two items are the default scope of the cookie, you will need to ensure that your cookie has the appropriate domain set. You will also want to set an expiration time for the cookie so that the browser knows to age it out.

Because the same user can login from multiple places or devices, you will need a separate session model to keep track of sessions. This model will likely be keyed on the session token, and will contain a reference to the user.

It is also fairly important that you ensure that the user has a reasonable experience when the session token expires. For example, since you are asking for JSON information and posting in a similar way, it is possible that a user will load the site, see their dates, then leave it open for a while before attempting to make a change. What will you do to ensure that something reasonable happens when your backend request (which is not tied to a page load!) fails due to a need to log in? How will you surface that to the user when the request is happening in the background?

One option would be to continually refresh the session in the background. Another would be to trigger a full reload to an error page if a background request fails because of an expired session.

Migrating Data

Once you have registered your first user (or a user that you want to have the original data), you should migrate the old data over to that user so that it is now behind a login. Make a migration plan and write it up. It can be a one-page document. Then execute the plan and verify that your data shows up for that user when logging in. Importantly, *also test that it does **not** show up for other users*.

Write-Up

For this lab, write up your migration plan. Discuss as part of the write-up what would need to happen (including how expiration of tokens would be handled, data migrated, etc.) if the following occurred:

- A user desires to change their username
- A user desires to change their password
- A user desires to delete their account and all associated data
- A user loses their password
- A user has their password stolen and used by someone else

Include what you learned. What surprised you about this task? What seems like it should be easier than it was? What seemed like it was easier than expected?

This write-up can be relatively short, but should be complete enough that someone technical could do the things you did.

Random Advice

As before, I will be posting random bits of advice on the lab, with code snippets, etc., as the week progresses. If you are stuck, ***make sure you use the labs channel to get help***. Also, ***help your peers***. I'm watching participation, and that's a *great* way of getting a good participation score. Be helpful, start conversations, collaborate. Remember that the only requirement that I have for lab copying is that you write your own code using the *ideas* that you share with others. Share ideas, share solutions, write your own code.