brevitas: a URL Shortener that Supports User Log-in & Link Management

Sandra Lee, Robyn Perry, Molly Robison, Sabina Shahbazzade

Introduction

brevitas is a link shortener website that allows users to shorten links, optionally log in, and manage and tag links.

User Login

Problem Being Solved

Allowing users to log in to *brevitas* stores the state of a particular user's interactions with our site.

Just as with many sites, a new user enters a username, email and password on our Create Account page. For security, we require passwords to be entered twice to ensure that the user has entered what they intended. Once the user chooses a password that passes the JavaScript validation, the username and password are sent to the server, the latter encrypted with the Python package bcrypt. The username is then checked against existing usernames to ensure uniqueness, then entered into the database along with the password hash and email address, and a success message is shown to the user once the Create Account page is re-rendered.

Existing users enter their username and password. We then ensure that the username exists and the password is correct. If this step fails, the page is re-rendered and the user is prompted to retry entering their credentials. Once they successfully log in, a session starts that allows their browser to keep a cookie with their credentials. The user can then see their history of shortened links and manage them.

Technology Details

Users who choose to log in to the site must enter valid credentials. We use JavaScript to enforce constraints for higher security around password composition, including length, upper and lowercase letters, and digits. We prevent duplicate usernames, duplicate emails, and using either username or email for password.

A function in our Python file does the checking of username against existing ones, and uses bcrypt to maintain the security of the password while checking that it's correct.

During account creation, if the password entered is not valid, the chosen username and email remain entered. This means the user doesn't have to re-enter those items, but needs only enter a stronger password.

We use a session cookie to store the currently logged in user's username, so our server can recognize that they're logged in.

After account creation, users sign in to access their account details. The app.py program checks whether the username exists and whether the password matches. Then, the database is queried using the username as unique identifier and the HTML table on the My Account page is populated with the set of links related to that user.

Problems Overcome

Because this process is pretty standard across websites, the challenge was mostly in implementing user account creation and log-in as smoothly as users will expect it to be. See the Future Improvements section for the additional features we would like to include.

Alternatives Considered & Tradeoffs

It would have been easier for implementation to not validate the password a second time, but this adds a level of feedback to the user so that they can be sure they typed in their password correctly. The password validation in general is optimized for security rather than ease for the user.

For password encryption, we opted for Python's bcrypt package instead of sha2, which we originally chose, for a couple of reasons. First of all, bcrypt generates and stores salts automatically, so we didn't need to worry if our salts were random enough, or about how they were being stored, and we weren't trying to build any security functionality that we don't know the best practices for. Secondly, sha2 is more easily broken than bcrypt (based on Blowfish), and while it seems unlikely that anyone will try to take down our password-shortening application, you can never be too careful.

We enforce uniqueness of usernames instead of relying on a hidden unique identifier in our database. This is efficient because it reduces storage costs in our database and utilizes username for a double purpose.

We opted to use borrowed code for email validation, but it is not the standard format for email validation known as RFC 822 as propagated by W3C. Its implementation required additional research and in the interest of getting a working prototype, we used something more expedient.

At first, we displayed Create Account fields in the middle of the home page. Later, we changed course to display the Create Account and My Account links in the upper right for consistency with other websites, and to reflect the fact that log-in is not a necessity. This removes visual clutter from the user's experience and provides a familiar path to site usage and log-in.

For usability's sake, we give feedback about whether a user has tried an incorrect username or password. This enhances a legitimate user's experience by helping them understand what they've botched in the login process. However, it is a security compromise: if someone is trying to break into another's account, they can get hints about what usernames are in the database.

One of the decisions we had to make was what to store in the cookie. We decided to store only username. This meant we could access the most up-to-date information for that user by querying our database instead of relying on data inside the cookie that could have been compromised in transmission.

Future Improvements

We would make improvements to the email validation code to comply with the internet standard RFC 822.

We would also like to do usability testing to make sure the interface maximizes ease and comfort to compensate for the high security password we require.

We would like to offer users who have forgotten their password the standard option of retrieving forgotten passwords or usernames.

Database Usage

Problem Being Solved

Using a database to store links allows a mapping between the short and long links created, and a mapping from them to the users, as well as to the tags they use to connect to these links.

Technology Details

We use a sqlite3 database to store the shortened and original links. We use a relational database with tables for URLs, Users, and Tags. The tables are related to each other using foreign keys so that all of the tables will relate to each other properly: the URL table references users, and the tags table references the short URLs in the URL table.

Our SQL statements are structured to protect against SQL injections. Queries to the database aren't reflected in the URL. Also, we pass variables to sqlite3's .execute function which automatically escapes the input. This adds greater security than concatenating strings to create an insert statement.

Problems Overcome

We ran into the issue of storing our database in our git repository. This caused a problem because whenever someone creates a new user account, the database was modified in git, and needed to be committed. Instead, we added our database to the .gitignore file. Alternately, we could have stored it in a separate directory and pointed our app.py file to it, but using the .gitignore file worked fine for our purposes and allowed us to navigate more efficiently when updating it while working on our project.

Alternatives Considered & Tradeoffs

Originally, we intended to store tags in the URL table. The problem with this would have been storing repeated rows in the URL table for each URL wherein all the data would have been replicated for each different tag.

For example, the URL table would have looked like so:

URL	short	username	date/time	times visited	tag1
URL	short	username	date/time	times visited	tag2
URL	short	username	date/time	times visited	tag3

It seemed cumbersome, and potentially problematic, to repeat data for each tag. We opted instead to create a new table for tags. This gives us the flexibility in the future to add more data related to tags, and eliminates duplicated data in the URL table.

Future Improvements

For greater security, we would also add server-side validation. In the current state of *brevitas*, all validation is done with JavaScript. However, we know that this is a vulnerability that malicious entities can exploit. JavaScript can be changed, so we would add functionality that prevents POST requests from submitting data into our database without SQL validation.

User History

Problem Being Solved

Keeping user history in the database gives us the opportunity to reflect their history back to them. It allows the users to store links they've shortened in the past, manage the links, and track the visits to those links. Users who get comfortable using our site will also benefit from being able to reuse frequently visited links that they've shortened rather than re-shortening. On the other hand, allowing them to shorten the same link into different short versions allows them to track visits to those links on different sites (for example, how many from Twitter clicked link A and how many from Facebook clicked link B?). This is helpful for those using shortened links for analytics.

We created a My Account page for all users that’s rendered once they’ve logged in with their credentials. This displays their history in an HTML table.

Furthermore, our site allows additional link management including sorting the table by column. This allows users to sort by date to see their most or least recent links, their short or long links (this helps with grouping URLs from a particular site), or by tags.

Perhaps most helpful are the analytics we provide. We track the number of times the short link redirects to the long link, so users can review how much traffic their shortened link has received.

A special feature that our site supports is IP lookup for users who choose not to log in. In the event that they later create an account, the links they’ve shortened will show up on the My Account page automatically. This allows users to quickly and easily use our site, and when they have the time to sign up for an account, their link shortening history will still be available to them.

Technology Details

To implement the basic feature of tracking number of clicks on the shortened link, we added a counter in the app.py function that maintains count each time the shortened link is visited.

For table sorting, we used Stupid Table, a jQuery table sorting plug-in.

In order to track user interactions when a user isn’t logged in, we collect IP address information that’s stored along shortened links created by whatever users are interacting with our site from that IP address. We access their address from the Flask request header.

The backend database and the table that we present to the user are completely composable: we could change how the database is implemented without having to change the table, and vice-versa. This allows us to display information in the table in a more user-intuitive way than it’s stored in the database, and to easily make changes to either one.

Problems Overcome

We wanted to show the tags for each link in a column of the My Account table. We set out trying to create a form within a form, but this isn’t permitted in HTML. Instead, we tried some workarounds using AJAX but didn’t have much success. What finally worked was keeping the single-form format. Upon submit, our Flask function checks whether there is tag data, and if there is, it’s treated as an insert into the database. If not, the action is treated like a delete submit that results in removed links. This accomplishes having a single form that allows execution of two different activities. Happily, you can both delete a link and tag another link at the same time.

Alternatives Considered & Tradeoffs

Because we have the IP lookup functionality that allows storing user interactions when a user is not logged in, we considered trying to match existing users with log-in credentials to those who have shortened links while not logged in. However, we decided that this strays from the "contract" a user makes with our website when they create an account. Most users wouldn't expect a site to be able to track their behavior while not signed in if they've previously created an account.

Furthermore, because we know that relying on IP address alone is faulty given that multiple users use the same machines sometimes, this seems more error-prone than it's worth. We certainly wouldn't want to implement a system that matches the wrong shortened links to a user's account just because their grandma used our link shortener while not logged in on their computer.

Future Improvements

First, we would add some visual cue that the table is sortable. The functionality is there but it is not clear that by clicking on the column headers produces a sorted table.

We would add functionality to delete tags, or deduplicate tags that have already been created.

Also, we'd like to implement tag deletion in such a way that it's decoupled from link deletion. Intuitively, it seems like it is a big deal to delete a link whose history you've been tracking, but a very minor deal to delete a tag you've just added. Independently of the actual back-end functionality, this intuitive difference should be reflected in what we make possible for our users.

We would add additional data and functionality related to tags. For example, we expect that users would like to be able to simply click on a tag and have only those entries appear that have that tag. Clicking the tag when it's in the sorted state would deselect that tag and all other entries would reappear.

We would also like to add more statistics related to managing these links. Perhaps it would be interesting to see a popularity metric, like what links have also been shortened by other users.

We would also hyperlink the short text shown in the table that grabs the full (not-so-short) shortened link so that users can easily click on the links they've shortened. This would help with user confidence. User verification that our site works is key to them wanting to continue using it!

Auto-shortening

Problem Being Solved

The auto-shortening allows users to quickly shorten links without having to specify the suffix that will be added to the shortened segment. The user can alternately choose a suffix, but is not obligated to. This offloads the job of thinking of a good ending for the shortened link from the user to our program, and immediately ensures a unique shortened link, avoiding the user experience of repeatedly trying suffixes that have already been taken.

We've also added functionality that removes any text entered in the "Shorten URL to" (we'll call this "user choice") field once the "Automatically create URL" button is clicked. This is useful feedback for a user and reduces confusion once the automatic option is selected.

Additionally, clicking in the user choice field changes the radio button selection automatically, reducing the number of clicks a user must make once they make up their mind to choose shorten text.

Technology Details

A user enters a long URL they'd like to shorten and then either chooses a set of characters as the suffix for the shortened link or lets us shorten it automatically for them. We constrain the length of randomly generated URLs to 6 characters, but user-generated suffixes can be any length but may not contain symbols. We ensured this with regular expression JavaScript validation.

Problems Overcome

While we were working on the Javascript that removes the text entered in the textbox when you select auto-shortening, we ran into a few issues. At first we were disabling the text box when the auto-generate radio button was selected, but we quickly realized that this precluded our other idea of automatically selecting the custom link radio button when you click into the custom link textbox. We finally decided that the most elegant way to deal with this was to erase text from the custom link box when the auto-generate radio button was selected, and to automatically select the custom link radio button when the custom link textbox gained focus. This prevented previous corner cases.

Alternatives Considered & Tradeoffs

One decision we considered was whether to check for conflicts in the database -- i.e. has anyone already chosen this shorten text? -- by making a single call to the database or by making a call for each auto-generated string we were checking. The tradeoff we were considering is performance versus accuracy: because we decided to make a single call each time a user (or our program) selects a short suffix, we are minimizing calls to the database but perhaps not accounting for other links being generated simultaneously. We opted for a single call because of TCP slow-start algorithms, so that we pull down a bunch of data at once rather than many small amounts of data.

If multiple users were shortening links simultaneously, there could be a collision of links. However, in our small implementation, this seemed like an acceptable route to take.

We also decided not to return an existing short link if a long link was entered twice into our system, because we figured that people would want to be able to see the statistics collected for only their short link, and not have their results skewed by others using our site.

Future Improvements

Perhaps we could implement a more semantically meaningful form of automatic shortening: rather than a random string of 6 characters, shorten it to a substring of the long link, or see if there's an available short link related to what other users have shortened the link to, or tags associated with the link.