Assignment: Blogz

In the last problem set we built a blog. We were able to persist data in the database and retrieve that data for display. However, there are still some serious feature gaps. We're going to fill those in now.

In this assignment, we'll refactor and expand our codebase to make this a multi-user blog site. We'll add authentication, so that users have to log in to create posts. Additionally, users will have their own blogs page, and visitors will be able to view blogs by author, or they can view all blog posts on the site by all authors. And we'll still maintain the ability to view an individual blog entry by itself.

Throughout the assignment, refer to the **demo app (http://blogz-demo.appspot.com/)** which models the functionality your app will have by the end of the assignment. Feel free to create an account, make a post or two, and poke around. Note that some of the urls won't match the ones that you have exactly, since we will be using query parameters which were not used in the demo app.

This assignment is the culmination of everything you've learned so far in Unit 2. Therefore, it is quite large and may seem like a lot of work. Just remember to take it one step at a time, and work on one task at a time. Some of these tasks we'll work through together, and some will be left to you to implement. Also, the video lessons for **Get It Done! (../../videos/get-it-done/)** will be invaluable as you work through this assignment. If you forget how to do something or encounter a bug, reviewing those lessons will help you figure it out!

Here are the steps we'll take as we build our impressive blog app:

Project Setup

Since this assignment is a continuation of *Build-a-Blog*, we'll want to use the files we created in that assignment, but we don't want to alter that repository. We want a separate repository called *Blogz*. To accomplish this, take the following steps:

1. At **GitHub.com (https://github.com/)**, create a new repository called `blogz`. Copy the URL for the repository.

   ⚠ **Warning**

   When creating the repository, be sure that you **do not** initialize it with a README, .gitignore, or license file. Doing so will make step 3 below more difficult.

2. In your terminal, `cd` into your `build-a-blog` repository. Make sure you are on your `master` branch and that it is up to date and contains your working solution to the last assignment.
3. Push this to the new repository you created on GitHub using the URL you copied: `git push https://github.com/yourUSERNAME/blogz.git`.
4. `cd` into your main development folder `LC101` (this will probably mean you just use `cd ..`).
5. Clone the repo locally using the same URL as you used to push: `git clone https://github.com/yourUSERNAME/blogz.git`.

Now you can `cd` into the `blogz` repo on your computer and start adding to and modifying it for this assignment!

✶ Note

> Be sure to activate your virtual environment once you `cd` into your new repo: `source activate flask-env`.

One very important change we still need to make is to the database. We'll want to make a new database specific to this assignment, and then alter our database connection string in `main.py` so that it uses this database (and the associated user/password). So follow the instructions for **creating a new database (../../studios/flicklist/6/#create-mysql-user-and-database)** using `blogz` as your user name and a password of your choice. Then change the connection string on this line of `main.py` so that it reflects this new database:

```python
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://blogz:password@localhost:8889/blogz'
```

Then you'll need to initialize your new database:

```
(flask-env) $ python
from main import db
db.create_all()
```

Overview of Improvements

This is a high-level overview of the major differences between your ***Build-a-Blog*** app and the ***Blogz*** app you will create in this assignment. We will tackle one change at a time; this just serves as a quick checklist that you can review to make sure you've added the necessary changes.

- We will add the following templates: `signup.html` , `login.html` , and `index.html` .
- We will also add a `singleUser.html` template that will be used to display only the blogs associated with a single given author. It will be used when we dynamically generate a page using a `GET` request with a `user` query parameter on the `/blog` route (similar to how we dynamically generated individual blog entry pages in the last assignment).
- We will add the following route handler functions: `signup` , `login` , and `index` .
- We'll have a `logout` function that handles a `POST` request to `/logout` and redirects the user to `/blog` after deleting the username from the session.
- We will also, naturally, have to add a `User` class to make all this new functionality possible, which is what we'll tackle next after a brief explanation of use cases.

A Note About Use Cases

Use cases help us break down our desired functionality into smaller, testable stories. Instead of just saying that we want our app to have a login feature, we further pare down what that means into example scenarios which we can then test one at a time. For instance, in this example we might break it down into the following three use cases:

1. User enters a username that is stored in the database with the correct password and is redirected to the `/newpost` page with their username being stored in a session.
2. User enters a username that is stored in the database with an incorrect password and is redirected to the `/login` page with a message that their password is incorrect.
3. User tries to login with a username that is not stored in the database and is redirected to the `/login` page with a message that this username does not exist.

This helps us define exactly what we want our app to do. It also makes it easier to break down our work into small, testable chunks. We will be using use cases informally in this assignment to make clear exactly what your app should do in different instances. If you want to learn more about how use cases are more formally used, read this **Usability.gov article (https://www.usability.gov/how-to-and-tools/methods/use-cases.html)**.

Add User Class

To get started, the first thing we'll want to do is add a `User` class and associated table to our app. It should have the following properties:

- `id` which is an `Integer` primary key that auto-increments (just like the others you've created in class)
- `username` which will be a `String` with a size of your choosing
- `password` which will also be a `String` with a size of your choosing
- `blogs` which signifies a relationship between the blog table and this user, thus binding this user with the blog posts they write.

✶ Note

> Feel free to reuse (with modifications) parts of the code from your *Get It Done!* app, or use **ours (https://github.com/LaunchCodeEducation/get-it-done/tree/d979a9991347431023d41abdd93891aedafc1f93)** since there are many similarities between that app and this one. Just make sure to modify it so that it matches the naming and functionality suggested in the assignment instructions on this page. And note that, as we mentioned in the video lessons, storing passwords directly in the database is **NOT** a good practice. We only do so in those video lessons and in this assignment because we have not yet covered hashing, which will be the subject of Class 13.

We'll also need to amend the `Blog` class in `main.py` (and in the database) so that it has a property called `owner_id` which is a foreign key linking the user's id to the blog post. And we'll need to amend the `Blog` constructor so that it takes in a user object (again, you can review the *Get It Done!* code for a reminder of how to do this). And think about what you'll need to do in your `/newpost` route handler function since there is a new parameter to consider when creating a blog entry.

After we make these changes to our code, we'll need to drop and re-create our tables using a Python shell. If you need a reminder of this process, review the second and third bullet items **here (../../studios/flicklist/6/#modify-flicklist-to-store-movie-ratings)**. Make sure to `from main import db, Blog, User` at the start of your shell session.

Add Login and Signup Pages

Let's make use of our new `User` class by enabling visitors to our web app to create an account and login. We'll need to make templates called `login.html` and `signup.html` and then create route handlers for them in `main.py` . With appropriate modifications, you'll be able to reuse a lot of the code you wrote for the ***Get It Done***. The code from your ***User Signup*** app may also come in handy for some of the validation tasks.

Make sure the following use cases are fulfilled:

- For `/login` page:

    - User enters a username that is stored in the database with the correct password and is redirected to the `/newpost` page with their username being stored in a session.
    - User enters a username that is stored in the database with an incorrect password and is redirected to the `/login` page with a message that their password is incorrect.
    - User tries to login with a username that is not stored in the database and is redirected to the `/login` page with a message that this username does not exist.
    - User does not have an account and clicks "Create Account" and is directed to the `/signup` page.

- For `/signup` page:

    - User enters new, valid username, a valid password, and verifies password correctly and is redirected to the '/newpost' page with their username being stored in a session.
    - User leaves any of the username, password, or verify fields blank and gets an error message that one or more fields are invalid.
    - User enters a username that already exists and gets an error message that username already exists.

- User enters different strings into the password and verify fields and gets an error message that the passwords do not match.
- User enters a password or username less than 3 characters long and gets either an invalid username or an invalid password message.

✶ **Note**

We leave it up to you whether to display error messages to the user using flash messages or by passing them to the template as you did in ***User Signup***. If you decide to do flash messages, add the necessary code to `main.py` and `base.html` .

Add Logout Function and Navigation

Now that users can login, we want to allow them to log out. To do so, you'll implement the same functionality you did in ***Get It Done*** and you'll add a link to your navigation in `base.html` with `href="/logout"` and a route handler function to `main.py` to handle that request. It should delete the username from the session and redirect to `/blog` .

While we're adding navigation links, let's also add a link to `"/login"` and to `"/"` , which will take users to the page we'll build in `index.html` that will display a list of all the usernames. You can call that page "Home".

✶ **Note**

Note that we have a `login.html` template and a `signup.html` template, but no `logout.html` template. It turns out that we don't need a template for logging out. Think about why this is the case. What makes logging out different from logging in, or creating an account?

Require Login

Now that we have a `User` class and the ability for visitors to signup and login, we don't want just anyone to be able to post on our blog site. We want to require that users have an account and be logged in to be able to access the `/newpost` page. So we'll need the `session` object to work this magic. And don't forget the secret key!!

And just as we did in the video lessons for *Get It Done*, we'll want to add a `require_login` function and decorate it with `@app.before_request`. The routes that we'll allow are: 'login', 'list_blogs' (or whatever you have named your route handler function for `/blog`), 'index', and 'signup'. We have not yet made the template or route handler for `/` and `index.html`, which corresponds to 'index' in the list above, but we'll add that shortly.

✶ Note

Note that when we use `request.endpoint` to match against our `allowed_routes` list, the `endpoint` is the name of the view *function*, not the url path. That is why in the list above we put `'login'` in the allowed_routes list, rather than `'/login'`. And if you have a different name for the view function for `/blog` than `list_blogs`, substitute your function name in the list we create above.

If the user is trying to go to any route besides these **and** is not logged in (their username is not stored in a session), then we want to redirect them to the `/login` page.

Functionality Check

At this point in the assignment, your app should also include the following functionality:

- User is logged in and adds a new blog post, then is redirected to a page featuring the individual blog entry they just created (as in *Build-a-Blog*).
- User visits the `/blog` page and sees a list of all blog entries by all users.
- User clicks on the title of a blog entry on the `/blog` page and lands on the individual blog entry page.
- User clicks "Logout" and is redirected to the `/blog` page and is unable to access the `/newpost` page (is redirected to `/login` page instead).

Make a Home Page

Now we can see a list of all blogs by all users on the `/blog` page, but what if a visitor to the site only wants to see the blogs for a particular author? To make that easy for the visitor, let's add a "Home" page that will live at the route `/` and will display a list of the usernames for all the authors on the site. Make a template called `index.html` that displays this list, and in `main.py` create a route handler function for it (named `index` so that it is included in the allowed routes we listed above).

Create Dynamic User Pages

Just as we created a page to dynamically display individual blog posts in **Build-a-Blog (../build-a-blog/#display-individual-entries)**, we'll create a page to dynamically display the posts of each individual user. We'll use a `GET` request on the `/blog` path with a query parameter of `?user=userId` where "userId" is the integer matching the id of the user whose posts we want to feature. And we'll need to create a template for this page.

There are three ways that users can reach this page and they all require that we make some changes to our templates. We will need to display, as a link, the username of the author of each blog post in a tagline on the individual blog entry page and on the `/blog` page. Check out our **demo app (http://blogz-demo.appspot.com/)** and see the line "Written by..." underneath the body of the blog posts.

Pro Tip

If you fulfilled the second bonus mission in **Build-a-Blog (../build-a-blog/#bonus-missions)** using a `DateTime` column, then you can utilize that field here to also note when each post was created, alongside the author.

✶ Note

Remember that each `Blog` object has an owner associated with it (passed to it in the constructor), so you can access the properties of that owner (such as `username` , or `id`) with dot notation.

Then you'll have to amend the `/blog` route handler to render the correct template (either the one for the individual blog user page, or the one for the individual blog entry page) based on the arguments in the request (i.e., which name the query parameter has). If the query param is `user` , then you need to use the template for the individual user page and pass it a list of all the blogs associated with that user.

We also need to modify our `index.html` . For each author name listed, add a link to the author's individual blog user page.

Here are the relevant use cases:

- User is on the `/` page ("Home" page) and clicks on an author's username in the list and lands on the individual blog user's page.
- User is on the `/blog` page and clicks on the author's username in the tagline and lands on the individual blog user's page.
- User is on the individual entry page (e.g., `/blog?id=1`) and clicks on the author's username in the tagline and lands on the individual blog user's page.

Bonus Missions

Before embarking on these bonus missions, make sure to commit and push your working code to GitHub!

Add Pagination

To limit the scrolling that users have to do if they visit a page with multiple blog posts on it, we'll implement pagination on our individual users page and our "all blogs" (`/blog`) page. The *Flask-SQLAlchemy* API makes this a fairly straightforward process. Review the documentation on pagination in the **Utilities (http://flask-sqlalchemy.pocoo.org/2.1/api/#utilities)** section and also the **Models (http://flask-sqlalchemy.pocoo.org/2.1/api/#models)**, which describes a `paginate` method that is part of the class `flask.ext.sqlalchemy.BaseQuery` .

We recommend limiting posts to 5 per page.

Add Hashing

After completing the video lessons on hashing, come back to this assignment and refactor your code so that you utilize hashing instead of storing passwords directly.

Submit

To turn in your assignment and get credit, follow the **submission instructions (../)**.