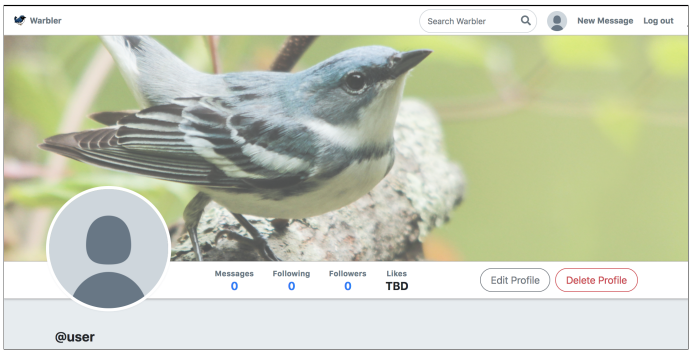
Warbler

Download starter code <../warbler.zip>

A Twitter clone with a Springboard spin!



<_images/warbler.png>

This exercise is intended to extend a somewhat-functioning Twitter clone. It is intended to give you practice reading and understanding an existing application, as well as fixing bugs in it, writing tests for it, and extending it with new features.

> **Warning: Clean & Commit As You Go**
>
> Treat this exercise like a real code base:
>
> - tidy and document as you go
> - check into Git often, as you add new features
>
> Practicing this process will help you as you work on larger codebases.
>
> Also, you may find bugs not mentioned in the app as you explore it. Fix these, and keep a log of what you fixed.

Setup

Create the Python virtual environment:

```
  :class: console

  $ python3 -m venv venv
  $ source venv/bin/activate
  (venv) $ pip install -r requirements.txt

Important Note:
If you are using Python 3.8 instead of 3.7, then you will have issues with installing some of the packages in the requirements.txt file into your virtual enviro
For Python 3.8 students, we recommend deleting pyscopg2-binary from the requirements.txt file, and using pip install pyscopg2-binary in the terminal in order to
```

Set up the database:

```
(venv) $ createdb warbler
(venv) $ python seed.py
```

Start the server:

```
(venv) $ flask run
```

Part One: Fix Current Features

Step One: Understand the Model

Read ***models.py***. Make a diagram of the four tables.

Note that the ***follows*** table has an unusual arrangement: it has two foreign keys to the same table. Why?

Using pdb, set a debugger in one of your routes, and hit it to pause code execution. From here, try out the various relationships set on the model classes. Get a feel for how they work.

~~Step Two: Fix Logout~~

~~Right now, there are links in the site to logout, but the logout route is not implemented.~~

~~On logout, it should flash a success message and redirect to the login page.~~

Step Three: Fix User Profile

~~The profile page for users works, but is missing a few things:~~

- ~~the location~~
- ~~the bio~~

- ~~the header image (which should be a background at the top)~~

~~Add these.~~

Step Four: Fix User Cards

~~On the followers, following, and list users pages, the user cards need to show the bio for the users. Add this.~~

Step Five: Profile Edit

There are buttons throughout the site for editing your profile, but this is unimplemented.

- ~~It should ensure a user is logged on (you can see how this is done in other routes)~~
- ~~It should show a form with the following:~~
  - ~~username~~
  - ~~email~~
  - ~~image_url~~
  - ~~header_image_url~~
  - ~~bio~~
  - ~~password *[see below]*~~
- ~~It should check that that password is the valid password for the user - if not, it should flash an error and return to the homepage.~~
- ~~It should edit the user for all of these fields *except* password (ie, this is not an area where users can change their passwords - the password is only for checking if it is the current correct password.~~
- ~~On success, it should redirect to the user detail page.~~

Step Six: Fix Homepage

The homepage for logged-in-users should show the last 100 warbles **only from the users ~~that~~ the logged-in user is following, and that user**, rather than warbles from *all* users.

Step Seven: Research and Understand Login Strategy

Look over the code in ***app.py*** related to authentication.

- How is the logged in user being kept track of?
- What is Flask's ***g*** object?
- What is the purpose of ***add_user_to_g***?
- ~~W~~hat does ***@app.before_request*** mean?

Part Two: Add Likes

> **Note: Do This Without AJAX/JavaScript**
>
> Eventually, this would be a fine feature to integrate with JS/AJAX — that's even a further study possibility!
>
> For now, though: please build this as a pure backend feature in Flask. Liking a warble should NOT be done via AJAX right now.

Add a new feature that allows a user to "like" a warble. They should only be able to like warbles written by other users. They should put a star (or some other similar symbol) next to liked warbles.

They should be able to unlike a warble, by clicking on that star.

On a profile page, it should show how many warblers that user has liked, and this should link to a page showing their liked warbles.

Part Three: Add Tests

Add tests. You'll need to proceed carefully here, since testing things like logging in and logging out will need to be tested using the session object.

Let's briefly discuss a couple of things related to tests: *how* you should test, and *what* you should test. We created some (mostly empty) test files:

- ***test_user_model.py***
- ***test_user_views.py***
- ***test_message_model.py***
- ***test_message_views.py***

In this case, there are four test files: two for testing the models, and two for testing the routes/view-functions.

We've put some boilerplate code into two of these to help you get started.

To run a file containing unittests, you can run the command `FLASK_ENV=production python -m unittest <name-of-python-file>`.

(we set ***FLASK_ENV*** for this command, so it doesn't use debug mode, and therefore won't use the Debug Toolbar during our tests).

If you are having an error running tests (comment out the line in your app.py that uses the Debug Toolbar)

So that's *how* you should test. But *what*, exactly, should you be testing? Let's take the above file structure as an example.

For model tests, you can simply verify that models have the attributes you expect, and write tests for any model methods.

Here are some questions your tests should answer for the **User** model:

1. Does the repr method work as expected?
2. Does ***is_following*** successfully detect when ***user1*** is following ***user2***?
3. Does ***is_following*** successfully detect when ***user1*** is not following ***user2***?
4. Does ***is_followed_by*** successfully detect when ***user1*** is followed by ***user2***?
5. Does ***is_followed_by*** successfully detect when ***user1*** is not followed by ***user2***?
6. Does ***User.create*** successfully create a new user given valid credentials?
7. Does ***User.create*** fail to create a new user if any of the validations (e.g. uniqueness, non-nullable fields) fail?
8. Does ***User.authenticate*** successfully return a user when given a valid username and password?
9. Does ***User.authenticate*** fail to return a user when the username is invalid?
10. Does ***User.authenticate*** fail to return a user when the password is invalid?

Try to formulate a similar set of questions for the **Message** model.

For the routing and view function tests, things get a bit more complicated. You should make sure that requests to all the endpoints supported in the ***views*** files return valid responses. Start by testing that the response code is what you expect, then do some light HTML testing to make sure the response is what you expect.

You should also be testing authentication and authorization. Here are some examples of questions your view function tests should answer regarding these ideas:

1. When you're logged in, can you see the follower / following pages for any user?
2. When you're logged out, are you disallowed from visiting a user's follower / following pages?
3. When you're logged in, can you add a message as yourself?
4. When you're logged in, can you delete a message as yourself?
5. When you're logged out, are you prohibited from adding messages?
6. When you're logged out, are you prohibited from deleting messages?
7. When you're logged in, are you prohibiting from adding a message as another user?
8. When you're logged in, are you prohibiting from deleting a message as another user?

(This isn't necessarily an exhaustive list of the tests you should write, but it should be enough to get you started.)

These tests are a bit trickier to write because they require you to make requests in the test, and look through the response in order to verify that the HTML you get back from the server looks correct.

Solution

Download Solution <solution>

Further Study

There are lots of areas of further study.

You won't have time to do all of these. Instead, pick those that seem most interesting to you.

Custom 404 Page

Learn how to add a custom 404 page, and make one.

Add AJAX

There are two areas where AJAX would really benefit this site:

- When you like/unlike a warble, you shouldn't have to refresh the page
- You should be able to compose a warble via a popup modal that is available on every page via the navigation bar button.

DRY Up the Templates

There's a lot of repetition in this app!

Here are some ideas to clean up repetition:

- Learn about the `{% include %}` statement in Jinja and use this to not have the forms be so repetitive.
- Learn about the `{% macro %}` and `{% import %}` statements in Jinja; you can use these to be even more clever, and get rid of a lot of repetition in the user detail, followers, followed_user pages, and more.

DRY Up the Authorization

Advanced but interesting

In many routes, there are a few lines that check for is-a-user-logged-in. You could solve this by writing your own "decorator", like "@app.route", but that checks if the ***g.user*** object is not null and, if not, flashes and redirects.

You'll need to do some searching and reading about Python decorators to do this.

DRY Up the URLs

Throughout the app, there are many, many places where URLs for the app are hardcoded throughout – consider the number of places that refer to URLs like ***/users/[user-id]***.

Flask has a nice feature, ***url_for()***, which can produce the correct URL when given a view function name. This allows you to not use the URLs directly in other routes/templates, and makes it easier in the future if you even needed to move URLs around (say, is ***/users/[user-id]*** needed to change to ***/users/detail/[user-id]***.

Learn about this feature and use it throughout the site.

Optimize Queries

In some places, Warbler may be making far more queries than it needs: the homepage can use more than 75 queries!

Using the Flask-DebugToolbar, audit query usage and fix some of the worst offenders.

Make a Change Password Form

Make a form with three fields:

- current password
- new password
- new password again, for confirmation

If the user is logged in *and* they provide the right password *and* their new passwords match, change their password.

Hint: do this by making a new method on the ***User*** class, rather than hard-coding stuff about password hashing in the view function.

Allow “Private” Accounts

Add a feature that allows a user to make their account “private”. A private account should normally only the profile page without messages.

You can follow a private account — but that user will need to approve your follow. At the point you are successfully following a private account, you should then be able to see their messages.

Note: this will require some schema changes and thoughtful design. Can you do this in a way that doesn’t sprinkle (even more) if conditions around? Can you add any useful functions on the ***User*** or ***Message*** classes?

Add Admin Users

Add a feature for “admin users” — these are users that have a new field on their model set to true.

Admin users can:

- delete any user’s messages
- delete any user
- edit a user profile; when an admin user edits a profile, they should be able to see and set the “admin” field to make another user an admin

User Blocking

Add a feature where users can block other users:

- when viewing a user page, there should be a block/unblock button
- blocked users view the blocker in any way

Direct Messages

Add a feature of “direct messages” — users being able to send private messages to another user, visible only to that user.

There are lots of possibilities on how far you want to take this one.