The Flask Mega-Tutorial Part VIII: Followers (/post/the-flask-mega-tutorial-part-viii-followers)

January 23 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Python (/category/Python), Database (/category/Database), Flask (/category/Flask), Programming (/category/Programming).

This is the eighth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to implement a "followers" feature similar to that of Twitter and other social networks.

For your reference, below is a list of the articles in this series.

- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

In this chapter I am going to work on the application's database some more. I want users of the application to be able to easily choose which other users they want to follow. So I'm going to be expanding the database so that it can keep track of who is following whom, which is harder than you may think.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.8), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.8.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.7...v0.8).

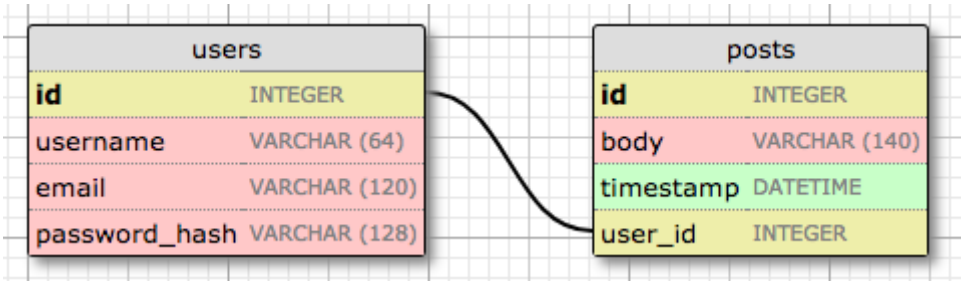
Database Relationships Revisited

I said above that I want to maintain a list of "followed" and "follower" users for each user. Unfortunately, a relational database does not have a list type that I can use for these lists, all there is are tables with records and relationships between these records.

The database has a table that represents users, so what's left is to come up with the proper relationship type that can model the follower/followed link. This is a good time to review the basic database relationship types:

One-to-Many

I have already used a one-to-many relationship in Chapter 4 (/post/the-flask-mega-tutorial-part-iv-database). Here is the diagram for this relationship:



The two entities linked by this relationship are users and posts. I say that a user has *many* posts, and a post has *one* user (or author). The relationship is represented in the database with the use of a *foreign key* on the "many" side. In the relationship above, the foreign key is the `user_id` field added to the `posts` table. This field links each post to the record of its author in the user table.

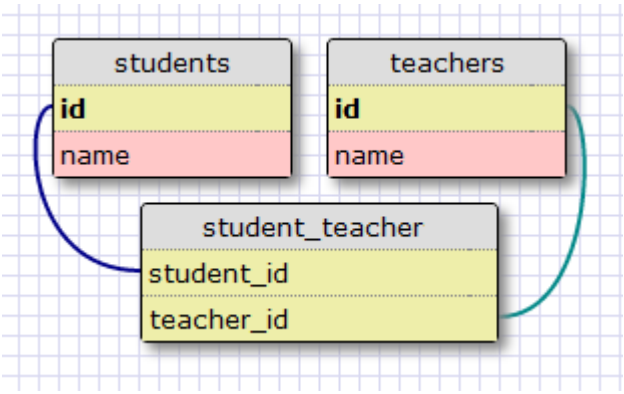
It is pretty clear that the `user_id` field provides direct access to the author of a given post, but what about the reverse direction? For the relationship to be useful I should be able to get the list of posts written by a given user. The `user_id` field in the `posts` table is also sufficient to answer this question, as databases have indexes that allow for efficient queries such us "retrieve all posts that have a user_id of X".

Many-to-Many

A many-to-many relationship is a bit more complex. As an example, consider a database that has `students` and `teachers`. I can say that a student has *many* teachers, and a teacher has *many* students. It's like two overlapped one-to-many relationships from both ends.

For a relationship of this type I should be able to query the database and obtain the list of teachers that teach a given student, and the list of students in a teacher's class. This is actually non-trivial to represent in a relational database, as it cannot be done by adding foreign keys to the existing tables.

The representation of a many-to-many relationship requires the use of an auxiliary table called an *association table*. Here is how the database would look for the students and teachers example:



While it may not seem obvious at first, the association table with its two foreign keys is able to efficiently answer all the queries about the relationship.

Many-to-One and One-to-One

A many-to-one is similar to a one-to-many relationship. The difference is that this relationship is looked at from the "many" side.

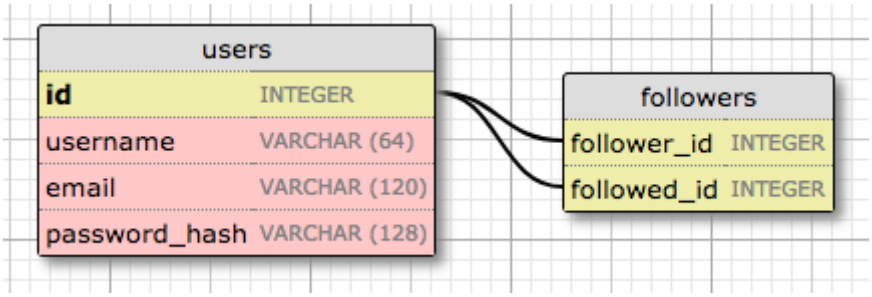
A one-to-one relationship is a special case of a one-to-many. The representation is similar, but a constraint is added to the database to prevent the "many" side to have more than one link. While there are cases in which this type of relationship is useful, it isn't as common as the other types.

Representing Followers

Looking at the summary of all the relationship types, it is easy to determine that the proper data model to track followers is the many-to-many relationship, because a user follows *many* users, and a user has *many* followers. But there is a twist. In the students and teachers example I had two entities that were related through the many-to-many relationship. But in the case of followers, I have users following other users, so there is just users. So what is the second entity of the many-to-many relationship?

The second entity of the relationship is also the users. A relationship in which instances of a class are linked to other instances of the same class is called a *self-referential relationship*, and that is exactly what I have here.

Here is a diagram of the self-referential many-to-many relationship that keeps track of followers:



The `followers` table is the association table of the relationship. The foreign keys in this table are both pointing at entries in the user table, since it is linking users to users. Each record in this table represents one link between a follower user and a followed user. Like the students and teachers example, a setup like this one allows the database to answer all the questions about followed and follower users that I will ever need. Pretty neat.

Database Model Representation

Let's add followers to the database first. Here is the `followers` association table:

app/models.py: Followers association table

```
followers = db.Table('followers',
    db.Column('follower_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('followed_id', db.Integer, db.ForeignKey('user.id'))
)
```

This is a direct translation of the association table from my diagram above. Note that I am not declaring this table as a model, like I did for the users and posts tables. Since this is an auxiliary table that has no data other than the foreign keys, I created it without an associated model class.

Now I can declare the many-to-many relationship in the users table:

app/models.py: Many-to-many followers relationship

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship(
        'User', secondary=followers,
        primaryjoin=(followers.c.follower_id == id),
        secondaryjoin=(followers.c.followed_id == id),
        backref=db.backref('followers', lazy='dynamic'), lazy='dynamic')
```

The setup of the relationship is non-trivial. Like I did for the `posts` one-to-many relationship, I'm using the `db.relationship` function to define the relationship in the model class. This relationship links `User` instances to other `User` instances, so as a

convention let's say that for a pair of users linked by this relationship, the left side user is following the right side user. I'm defining the relationship as seen from the left side user with the name `followed`, because when I query this relationship from the left side I will get the list of followed users (i.e those on the right side). Let's examine all the arguments to the `db.relationship()` call one by one:

- `'User'` is the right side entity of the relationship (the left side entity is the parent class). Since this is a self-referential relationship, I have to use the same class on both sides.
- `secondary` configures the association table that is used for this relationship, which I defined right above this class.
- `primaryjoin` indicates the condition that links the left side entity (the follower user) with the association table. The join condition for the left side of the relationship is the user ID matching the `follower_id` field of the association table. The `followers.c.follower_id` expression references the `follower_id` column of the association table.
- `secondaryjoin` indicates the condition that links the right side entity (the followed user) with the association table. This condition is similar to the one for `primaryjoin`, with the only difference that now I'm using `followed_id`, which is the other foreign key in the association table.
- `backref` defines how this relationship will be accessed from the right side entity. From the left side, the relationship is named `followed`, so from the right side I am going to use the name `followers` to represent all the left side users that are linked to the target user in the right side. The additional `lazy` argument indicates the execution mode for this query. A mode of `dynamic` sets up the query to not run until specifically requested, which is also how I set up the posts one-to-many relationship.
- `lazy` is similar to the parameter of the same name in the `backref`, but this one applies to the left side query instead of the right side.

Don't worry if this is hard to understand. I will show you how to work with these queries in a moment, and then everything will become clearer.

The changes to the database need to be recorded in a new database migration:

```
(venv) $ flask db migrate -m "followers"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'followers'
  Generating /home/miguel/microblog/migrations/versions/ae346256b650_followers.py ... done

(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 37f06a334dbf -> ae346256b650, followers
```

Adding and Removing "follows"

Thanks to the SQLAlchemy ORM, a user following another user can be recorded in the database working with the `followed` relationship as if it was a list. For example, if I had two users stored in `user1` and `user2` variables, I can make the first follow the second with this simple statement:

```
user1.followed.append(user2)
```

To stop following the user, then I could do:

```
user1.followed.remove(user2)
```

Even though adding and removing followers is fairly easy, I want to promote reusability in my code, so I'm not going to sprinkle "appends" and "removes" through the code. Instead, I'm going to implement the "follow" and "unfollow" functionality as methods in the `User` model. It is always best to move the application logic away from view functions and into models or other auxiliary classes or modules, because as you will see later in this chapter, that makes unit testing much easier.

Below are the changes in the user model to add and remove relationships:

app/models.py: Add and remove followers

```
class User(UserMixin, db.Model):
    #...

    def follow(self, user):
        if not self.is_following(user):
            self.followed.append(user)

    def unfollow(self, user):
        if self.is_following(user):
            self.followed.remove(user)

    def is_following(self, user):
        return self.followed.filter(
            followers.c.followed_id == user.id).count() > 0
```

The `follow()` and `unfollow()` methods use the `append()` and `remove()` methods of the relationship object as I have shown above, but before they touch the relationship they use the `is_following()` supporting method to make sure the requested action makes sense. For example, if I ask `user1` to follow `user2`, but it turns out that this following relationship already exists in the database, I do not want to add a duplicate. The same logic can be applied to unfollowing.

The `is_following()` method issues a query on the `followed` relationship to check if a link between two users already exists. You have seen me use the `filter_by()` method of the SQLAlchemy query object before, for example to find a user given its username. The `filter()` method that I'm using here is similar, but lower level, as it can include arbitrary filtering conditions, unlike `filter_by()` which can only check for equality to a constant value. The condition that I'm using in `is_following()` looks for items in the association table that have the left side foreign key set to the `self` user, and the right side set to the `user` argument. The query is terminated with a `count()` method, which returns the number of results. The result of this query is going to be `0` or `1`, so checking for the count being 1 or greater than 0 is actually equivalent. Other query terminators you have seen me use in the past are `all()` and `first()`.

Obtaining the Posts from Followed Users

Support for followers in the database is almost complete, but I'm actually missing one important feature. In the index page of the application I'm going to show blog posts written by all the people that are followed by the logged in user, so I need to come up with a database query that returns these posts.

The most obvious solution is to run a query that returns the list of followed users, which as you already know, it would be `user.followed.all()`. Then for each of these returned users I can run a query to get the posts. Once I have all the posts I can merge them into a single list and sort them by date. Sounds good? Well, not really.

This approach has a couple of problems. What happens if a user is following a thousand people? I would need to execute a thousand database queries just to collect all the posts. And then I will need to merge and sort the thousand lists in memory. As a secondary problem, consider that the application's home page will eventually have *pagination* implemented, so it will not display all the available posts but just the first few, with a link to get more if desired. If I'm going to display posts sorted by their date, how can I know which posts are the most recent of all followed users combined, unless I get all the posts and sort them first? This is actually an awful solution that does not scale well.

There is really no way to avoid this merging and sorting of blog posts, but doing it in the application results in a very inefficient process. This kind of work is what relational databases excel at. The database has indexes that allow it to perform the queries and the sorting in a much more efficient way that I can possibly do from my side. So what I really want is to come up with a single database query that defines the information that I want to get, and then let the database figure out how to extract that information in the most efficient way.

Below you can see this query:

app/models.py: Followed posts query

```
class User(UserMixin, db.Model):
    #...
    def followed_posts(self):
        return Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)).filter(
                followers.c.follower_id == self.id).order_by(
                    Post.timestamp.desc())
```

This is by far the most complex query I have used on this application. I'm going to try to decipher this query one piece at a time. If you look at the structure of this query, you are going to notice that there are three main sections designed by the `join()`, `filter()` and `order_by()` methods of the SQLAlchemy query object:

```
Post.query.join(...).filter(...).order_by(...)
```

Joins

To understand what a join operation does, let's look at an example. Let's assume that I have a `User` table with the following contents:

id	username
1	john
2	susan
3	mary
4	david

To keep things simple I am not showing all the fields in the user model, just the ones that are important for this query.

Let's say that the `followers` association table says that user `john` is following users `susan` and `david`, user `susan` is following `mary` and user `mary` is following `david`. The data that represents the above is this:

follower_id	followed_id
1	2
1	4
2	3
3	4

Finally, the posts table contains one post from each user:

id	text	user_id
1	post from susan	2
2	post from mary	3
3	post from david	4
4	post from john	1

This table also omits some fields that are not part of this discussion.

Here is the `join()` call that I defined for this query once again:

```
Post.query.join(followers, (followers.c.followed_id == Post.user_id))
```

I'm invoking the join operation on the posts table. The first argument is the followers association table, and the second argument is the join *condition*. What I'm saying with this call is that I want the database to create a temporary table that combines data from posts and followers tables. The data is going to be merged according to the condition that I passed as argument.

The condition that I used says that the `followed_id` field of the followers table must be equal to the `user_id` of the posts table. To perform this merge, the database will take each record from the posts table (the left side of the join) and append any records from the `followers` table (the right side of the join) that match the condition. If multiple records in `followers` match the condition, then the post entry will be repeated for each. If for a given post there is no match in followers, then that post record is not part of the join.

With the example data I defined above, the result of the join operation is:

id	text	user_id	follower_id	followed_id
1	post from susan	2	1	2
2	post from mary	3	2	3
3	post from david	4	1	4
3	post from david	4	3	4

Note how the `user_id` and `followed_id` columns are equal in all cases, as this was the join condition. The post from user `john` does not appear in the joined table because there are no entries in followers that have `john` as a followed user, or in other words, nobody is following john. And the post from `david` appears twice, because that user is followed by two different users.

It may not be immediately clear what do I gain by creating this join, but keep reading, as this is just one part of the bigger query.

Filters

The join operation gave me a list of all the posts that are followed by some user, which is a lot more data that I really want. I'm only interested in a subset of this list, the posts followed by a single user, so I need trim all the entries I don't need, which I can do with a `filter()` call.

Here is the filter portion of the query:

```
filter(followers.c.follower_id == self.id)
```

Since this query is in a method of class `User`, the `self.id` expression refers to the user ID of the user I'm interested in. The `filter()` call selects the items in the joined table that have the `follower_id` column set to this user, which in other words means that I'm keeping only the entries that have this user as a follower.

Let's say the user I'm interested in is `john`, which has its `id` field set to 1. Here is how the joined table looks after the filtering:

id	text	user_id	follower_id	followed_id
1	post from susan	2	1	2
3	post from david	4	1	4

And these are exactly the posts that I wanted!

Remember that the query was issued on the `Post` class, so even though I ended up with a temporary table that was created by the database as part of this query, the result will be the posts that are included in this temporary table, without the extra columns added by the join operation.

Sorting

The final step of the process is to sort the results. The part of the query that does that says:

```
order_by(Post.timestamp.desc())
```

Here I'm saying that I want the results sorted by the timestamp field of the post in descending order. With this ordering, the first result will be the most recent blog post.

Combining Own and Followed Posts

The query that I'm using in the `followed_posts()` function is extremely useful, but has one limitation. People expect to see their own posts included in their timeline of followed users, and the query as it is does not have that capability.

There are two possible ways to expand this query to include the user's own posts. The most straightforward way is to leave the query as it is, but make sure all users are following themselves. If you are your own follower, then the query as shown above will find your own posts along with those of all the people you follow. The disadvantage of this method is that it affects the stats regarding followers. All follower counts are going to be inflated by one, so they'll have to be adjusted before they are shown. The second way to do this is by create a second query that returns the user's own posts, and then use the "union" operator to combine the two queries into a single one.

After considering both options I decided to go with the second one. Below you can see the `followed_posts()` function after it has been expanded to include the user's posts through a union:

app/models.py: Followed posts query with user's own posts.

```
    def followed_posts(self):
        followed = Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)).filter(
                followers.c.follower_id == self.id)
        own = Post.query.filter_by(user_id=self.id)
        return followed.union(own).order_by(Post.timestamp.desc())
```

Note how the `followed` and `own` queries are combined into one, before the sorting is applied.

Unit Testing the User Model

While I don't consider the followers implementation I have built a "complex" feature, I think it is also not trivial. My concern when I write non-trivial code, is to ensure that this code will continue to work in the future, as I make modifications on different parts of the application. The best way to ensure that code you have already written continues to work in the future is to create a suite of automated tests that you can re-run each time changes are made.

Python includes a very useful `unittest` package that makes it easy to write and execute unit tests. Let's write some unit tests for the existing methods in the `User` class in a *tests.py* module:

tests.py: User model unit tests.

```
from datetime import datetime, timedelta
import unittest
from app import app, db
from app.models import User, Post

class UserModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_password_hashing(self):
        u = User(username='susan')
        u.set_password('cat')
        self.assertFalse(u.check_password('dog'))
        self.assertTrue(u.check_password('cat'))

    def test_avatar(self):
        u = User(username='john', email='john@example.com')
        self.assertEqual(u.avatar(128), ('https://www.gravatar.com/avatar/'
                                         'd4c74594d841139328695756648b6bd6'
                                         '?d=identicon&s=128'))

    def test_follow(self):
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        db.session.add(u1)
        db.session.add(u2)
        db.session.commit()
        self.assertEqual(u1.followed.all(), [])
        self.assertEqual(u1.followers.all(), [])

        u1.follow(u2)
        db.session.commit()
        self.assertTrue(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 1)
        self.assertEqual(u1.followed.first().username, 'susan')
        self.assertEqual(u2.followers.count(), 1)
        self.assertEqual(u2.followers.first().username, 'john')

        u1.unfollow(u2)
        db.session.commit()
        self.assertFalse(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 0)
        self.assertEqual(u2.followers.count(), 0)

    def test_follow_posts(self):
        # create four users
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        u3 = User(username='mary', email='mary@example.com')
        u4 = User(username='david', email='david@example.com')
        db.session.add_all([u1, u2, u3, u4])
```

```
        # create four posts
        now = datetime.utcnow()
        p1 = Post(body="post from john", author=u1,
                  timestamp=now + timedelta(seconds=1))
        p2 = Post(body="post from susan", author=u2,
                  timestamp=now + timedelta(seconds=4))
        p3 = Post(body="post from mary", author=u3,
                  timestamp=now + timedelta(seconds=3))
        p4 = Post(body="post from david", author=u4,
                  timestamp=now + timedelta(seconds=2))
        db.session.add_all([p1, p2, p3, p4])
        db.session.commit()

        # setup the followers
        u1.follow(u2)  # john follows susan
        u1.follow(u4)  # john follows david
        u2.follow(u3)  # susan follows mary
        u3.follow(u4)  # mary follows david
        db.session.commit()

        # check the followed posts of each user
        f1 = u1.followed_posts().all()
        f2 = u2.followed_posts().all()
        f3 = u3.followed_posts().all()
        f4 = u4.followed_posts().all()
        self.assertEqual(f1, [p2, p4, p1])
        self.assertEqual(f2, [p2, p3])
        self.assertEqual(f3, [p3, p4])
        self.assertEqual(f4, [p4])

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

I have added four tests that exercise the password hashing, user avatar and followers functionality in the user model. The `setUp()` and `tearDown()` methods are special methods that the unit testing framework executes before and after each test respectively. I have implemented a little hack in `setUp()`, to prevent the unit tests from using the regular database that I use for development. By changing the application configuration to `sqlite://` I get SQLAlchemy to use an in-memory SQLite database during the tests. The `db.create_all()` call creates all the database tables. This is a quick way to create a database from scratch that is useful for testing. For development and production use I have already shown you how to create database tables through database migrations.

You can run the entire test suite with the following command:

```
(venv) $ python tests.py
test_avatar (__main__.UserModelCase) ... ok
test_follow (__main__.UserModelCase) ... ok
test_follow_posts (__main__.UserModelCase) ... ok
test_password_hashing (__main__.UserModelCase) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.494s

OK
```

From now on, every time a change is made to the application, you can re-run the tests to make sure the features that are being tested have not been affected. Also, each time another feature is added to the application, a unit test should be written for it.

Integrating Followers with the Application

The support of followers in the database and models is now complete, but I don't have any of this functionality incorporated into the application, so I'm going to add that now.

Because the follow and unfollow actions introduce changes in the application, I'm going to implement them as `POST` requests, which are triggered from the web browser as a result of submitting a web form. It would be easier to implement these routes as `GET` requests, but then they could be exploited in CSRF (http://en.wikipedia.org/wiki/Cross-site_request_forgery) attacks. Because `GET` requests are harder to protect against CSRF, they should only be used on actions that do not introduce state changes. Implementing these as a result of a form submission is better because then a CSRF token can be added to the form.

But how can a follow or unfollow action be triggered from a web form when the only thing the user needs to do is click on "Follow" or "Unfollow", without submitting any data? To make this work, the form is not going to have any data fields. The only elements in the form are going to be the CSRF token, which is implemented as a hidden field and added automatically by Flast-WTF, and a submit button, which is going to be what the user needs to click to trigger the action. Since the two actions are almost identical I'm going to use the same form for both. I'm going to call this form `EmptyForm`.

app/forms.py: Empty form for following and unfollowing.

```
class EmptyForm(FlaskForm):
    submit = SubmitField('Submit')
```

Let's add two new routes in the application to follow and unfollow a user:

app/routes.py: Follow and unfollow routes.

```
from app.forms import EmptyForm

# ...

@app.route('/follow/<username>', methods=['POST'])
@login_required
def follow(username):
    form = EmptyForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=username).first()
        if user is None:
            flash('User {} not found.'.format(username))
            return redirect(url_for('index'))
        if user == current_user:
            flash('You cannot follow yourself!')
            return redirect(url_for('user', username=username))
        current_user.follow(user)
        db.session.commit()
        flash('You are following {}!'.format(username))
        return redirect(url_for('user', username=username))
    else:
        return redirect(url_for('index'))


@app.route('/unfollow/<username>', methods=['POST'])
@login_required
def unfollow(username):
    form = EmptyForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=username).first()
        if user is None:
            flash('User {} not found.'.format(username))
            return redirect(url_for('index'))
        if user == current_user:
            flash('You cannot unfollow yourself!')
            return redirect(url_for('user', username=username))
        current_user.unfollow(user)
        db.session.commit()
        flash('You are not following {}.'.format(username))
        return redirect(url_for('user', username=username))
    else:
        return redirect(url_for('index'))
```

The form handling in these routes is simpler, because we only have to implement the submission part. Unlike other forms such as the login and edit profile forms, these two forms do not have their own pages, the forms will be rendered by the `user()` route and will appear in the user's profile page. The only reason why the `validate_on_submit()` call can fail is if the CSRF token is missing or invalid, so in that case I just redirect the application back to the home page.

If the form validation passes, I do some error checking before actually carrying out the follow or unfollow action. This is to prevent unexpected issues, and to try to provide a useful message to the user when a problem has occurred.

To render the follow or unfollow button, I need to instantiate an `EmptyForm` object and pass it to the *user.html* template. Because these two actions are mutually exclusive, I can pass a single instance of this generic form to the template:

app/routes.py: Follow and unfollow routes.

```
@app.route('/user/<username>')
@login_required
def user(username):
    # ...
    form = EmptyForm()
    return render_template('user.html', user=user, posts=posts, form=form)
```

I can now add the follow or unfollow forms in the profile page of each user:

app/templates/user.html: Follow and unfollow links in user profile page.

```
        ...
        <h1>User: {{ user.username }}</h1>
        {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
        {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p>{% endif %}
        <p>{{ user.followers.count() }} followers, {{ user.followed.count() }} following.</p>
        {% if user == current_user %}
        <p><a href="{{ url_for('edit_profile') }}">Edit your profile</a></p>
        {% elif not current_user.is_following(user) %}
        <p>
            <form action="{{ url_for('follow', username=user.username) }}" method="post">
                {{ form.hidden_tag() }}
                {{ form.submit(value='Follow') }}
            </form>
        </p>
        {% else %}
        <p>
            <form action="{{ url_for('unfollow', username=user.username) }}" method="post">
                {{ form.hidden_tag() }}
                {{ form.submit(value='Unfollow') }}
            </form>
        </p>
        {% endif %}
        ...
```

The changes to the user profile template add a line below the last seen timestamp that shows how many followers and followed users this user has. And the line that has the "Edit" link when you are viewing your own profile now can have one of three possible links:

- If the user is viewing his or her own profile, the "Edit" link shows as before.
- If the user is viewing a user that is not currently followed, the "Follow" form shows.
- If the user is viewing a user that is currently followed, the "Unfollow" form shows.

To reuse the `EmptyForm()` instance for both the follow and unfollow forms, I pass a `value` argument when rendering the submit button. In a submit button, the `value` attribute defines the label, so with this trick I can change the text in the submit button depending on the action that I need to present to the user.

At this point you can run the application, create a few users and play with following and unfollowing users. The only thing you need to remember is to type the profile page URL of the user you want to follow or unfollow, since there is currently no way to see a list of users. For example, if you want to follow a user with the `susan` username, you will need to type *http://localhost:5000/user/susan* in the browser's address bar to access the profile page for that user. Make sure you check how the followed and follower user counts change as you issue follows or unfollows.

I should be showing the list of followed posts in the index page of the application, but I don't have all the pieces in place to do that yet, since users cannot write blog posts yet. So I'm going to delay this change until that functionality is in place.