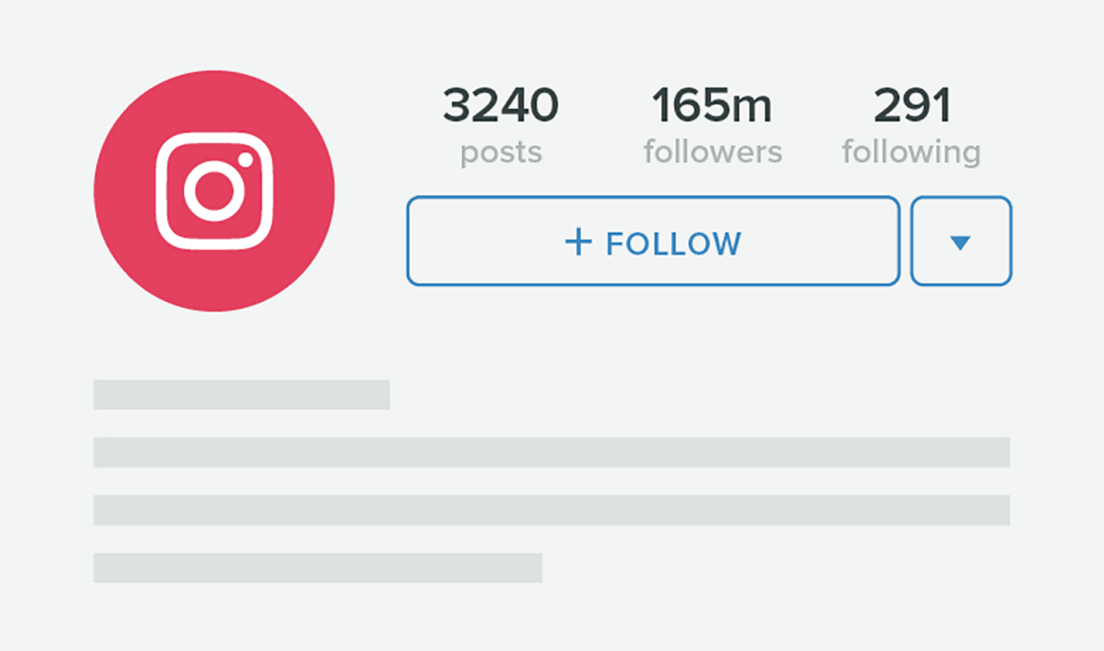
E-14a

Lab 8: Implement “followers” feature



Follow Feature on Instagram

Learning Objectives

- More complex database manipulation
- Query database
- Understand and implement CRUD
- Understand /profile/<username>
- Login users see all posts him/herself made
- Login users to see all his/her following users’ posts
- Deploy using Heroku

INTRODUCTION

Socially aware Web applications allow users to connect with other users. Applications call these relationships followers, friends, contacts, connections, or buddies, but the feature is the same regardless of the name, and in all cases involves keeping track of directional links between pairs of users and using these links in database queries.

source : Miguel Grinberg, "Flask Web Development"

In this lab we will learn how to implement a follower feature in Flask. Users will be able to "follow" other users and choose to filter the post list on the home page to include only those from the users they follow.

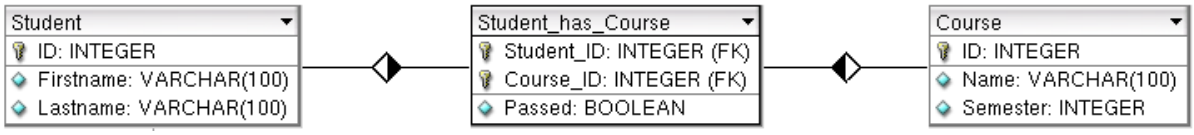
Database Relationships

As discussed earlier, databases establish links between records using relationships. The *one-to-many* relationship is the most common type of relationship, where a record is linked with a list of related records. Most other relationship types can be derived from the *one-to-many* type. The *one-to-one* relationship type is a simplification of the *one-to-many*, where "many" side is constrained to only have at most one element. The only relationship type that cannot be implemented as a simple variation of the *one-to-many* model is the ***many-to-many***, which has lists of elements on both sides. This relationship will be used in Lab 8.

Many-to-many Relationships

Remember from Lecture 4, a classical example of a *many-to-many* relationships would be a database of students and the classes they are taking. Given this setting, it is impossible to add a foreign key to a class in the students table, because a student takes many classes. Likewise, you cannot add a foreign key to the student in the classes table, because classes have more than one student.

The solution is to add a third table to the database, called an *association table*. Now the *many-to-many* relationships can be decomposed into two *one-to-many* relationships from each of the two original tables to the association table. Next figure shows how the *many-to-many* relationships between students and classes is represented:

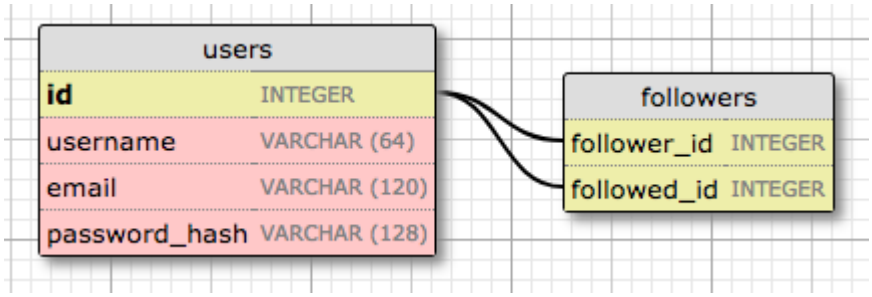


Many-to-many relationships

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* relationships, because a user follows many users, and a user has many followers. But there is a twist. In the students and courses example we had two entities that were related through the *many-to-many* relationships. But in the case of followers, we have users following other users, so there is just users.

The second entity of the relationship should also be users. A relationship in which instances of a class are linked to other instances of the same class is called a *self-referential relationship*.



source: the-flask-mega-tutorial-part-viii-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 course example, a setup like this one allows the database to answer all the questions about followed and follower users.

Database "lab_8"

Let us implement the previously described scenario. Open Postgres and create new database with name “lab8_db”. The database should contain 3 tables: “users”, “posts” and “follows”:

```
CREATE TABLE users (
      uid serial NOT NULL PRIMARY KEY,
      username TEXT NOT NULL,
      password TEXT NOT NULL
      );

CREATE TABLE posts (
      pid serial NOT NULL PRIMARY KEY,
      author serial NOT NULL,
      content TEXT NOT NULL,
      FOREIGN KEY (author) REFERENCES users(uid)
      );

CREATE TABLE follows (
      fid serial NOT NULL PRIMARY KEY,
      follower serial NOT NULL,
      following serial NOT NULL,
      FOREIGN KEY (follower) REFERENCES users(uid),
      FOREIGN KEY (following) REFERENCES users(uid)
      );
```

Populate the previously created tables:

```
INSERT INTO users (username, password) VALUES ('test_user1', 'test_pw1');
INSERT INTO users (username, password) VALUES ('test_user2', 'test_pw2');
INSERT INTO users (username, password) VALUES ('e14a', '123');

INSERT INTO posts (author, content) VALUES (1, 'Post user 1');
INSERT INTO posts (author, content) VALUES (2, 'Post user 2');
INSERT INTO posts (author, content) VALUES (3, 'E14a Hello Post');

INSERT INTO follows (follower, following) VALUES (1, 2);
INSERT INTO follows (follower, following) VALUES (2, 1);
```

Search Users and Profile Pages

In order to start following users, we should be able to search for them first. Let us create the *profile.html*

page and profile route.

In the *profile.html*, we have to create a return to Home button:

```
<h5><a href="{{ url_for('index') }}">Home</a></h5>
```

List all posts:

```
{% for p in posts %}
    <p>Author@{{ p.author }}: {{ p.content }}</p>
{% endfor %}
```

In *routes.py* create new profile route:

```
@app.route('/profile/<username>', methods=['GET'])
def profile(username):
    profile_user = User.query.filter_by(username=username).first()
    profile_user_posts = Post.query.filter_by(author=profile_user.uid).all()

    if "username" in session:
      session_user = User.query.filter_by(username=session['username']).first()
      return render_template('profile.html',
      user=profile_user, posts=profile_user_posts)
```

The `@app.route` decorator to declare this view function looks a little bit different than the previous ones. In this case we have a dynamic component in it, which is indicated as the `<username>` URL component that is surrounded by < and >. When a route has a dynamic component, Flask will accept any text in that portion of the URL, and will invoke the view function with the actual text as an argument. For example, if the client browser requests URL `/user/susan`, the view function is going to be called with the argument username set to 'susan'.

Next to implement is Search functionality. Search component includes the creation of html elements and adding a new route. Open up the *index.html* and place the following code at the bottom of the file:

```
<form action="/search" method="POST">
    <input type="text" name="search_box"></input>
    <button type="submit"> Search </button>
</form>
```

These lines define the search bar (input type) and a Search button. Now, let us create a new route in routes.py:

```
@app.route('/search', methods=['POST'])
def search():
      user_to_query = request.form['search_box']
      return redirect(url_for('profile', username=user_to_query))
```

Followers

First, we have to define the “follower” and “following” attributes in database model representation. In *models.py* under basic attributes, place these lines of code:

```
class Follows(db.Model):
   __tablename__ = 'follows'
   fid    =    db.Column(db.Integer,    primary_key=True,    autoincrement=True)
   follower = db.Column(db.Integer, db.ForeignKey('users.uid'), nullable=False)
   following = db.Column(db.Integer, db.ForeignKey('users.uid'), nullable=False)
```

These correspond to a DB schema previously defined. Both, “follower” and “following” attributes have “uid” as a foreign key.

Follow/Unfollow Functionality

We are 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.

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

```
def is_following(self, user):
      return self.followed.filter(follows.c.following == user.uid).count() > 0

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)
```

The `follow()` and `unfollow()` methods use the `append()` and `remove()` methods of the relationship object. But, before they touch the relationship they use the `is_following()` supporting method to make sure the requested action makes sense.

The `is_following()` method issues a query on the followed relationship to check if a link between two users already exists. The `filter_by()` method has been used here to include arbitrary filtering conditions, which can only check for equality to a constant value. The condition that we are using here is the `is_following()`, which 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 are `all()` and `first()`.

The next step is to run a query that returns the list of followed users. Then for each of these returned users we can run a query to get the posts. Open up your *models.py* and add:

```
def followed_posts(self):
   followed_posts = Post.query.join(follows,
   (follows.c.following == Post.author)).filter(follows.c.follower
   == self.uid)
      self_posts = Post.query.filter_by(author=self.uid)
      return followed_posts.union(self_posts)
```

If you look at the structure of this query, you are going to notice that there are 2 main sections designed by the `join()` and `filter()` methods. We are invoking the join operation on the posts table. The first argument is the followers association table, and the second argument is the join condition. We 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 we passed as argument.

The condition that I used says that the `followed_posts` 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.

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_posts`

column set to this user, which in other words means that I'm keeping only the entries that have this user as a follower.

Integrating Followers with the Application

The support followers in the database and models is now complete, but we don't have any of this functionality incorporated into the application. Define follow/unfollow buttons in *profile.hml*:

```
{% if followed is defined %}
        {% if followed %}
        <form action="/unfollow/{{user.username}}" method="POST">
            <button type="submit"> Unfollow </button>
            </form>
        {% else %}
            <form action="/follow/{{user.username}}" method="POST">
            <button type="submit"> Follow </button>
            </form>
        {% endif %}
{% endif %}
```

Update profiles route with (**bold**):

```
if "username" in session:
    session_user =User.query.filter_by(username=session['username']).first()
    if Follows.query.filter_by(follower=session_user.uid,
    following=profile_user.uid).first():
        followed = True

    else:
        followed = False

    return render_template('profile.html', user=profile_user,
    posts=profile_user_posts, followed=followed)
```

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

```
# Follow route
@app.route('/follow/<username>', methods=['POST'])
def follow(username):
    session_user = User.query.filter_by(username=session['username']).first()
```

```
    user_to_follow = User.query.filter_by(username=username).first()
    new_follow =  Follows(follower=session_user.uid,
    following=user_to_follow.uid)

    db.session.add(new_follow)
    db.session.commit()
    return redirect(url_for('profile', username=username))

# Unfollow route
@app.route('/unfollow/<username>', methods=['POST'])
def unfollow(username):
    session_user = User.query.filter_by(username=session['username']).first()
    user_to_unfollow = User.query.filter_by(username=username).first()

    delete_follow = Follows.query.filter_by(follower=session_user.uid,
    following=user_to_unfollow.uid).first()
    db.session.delete(delete_follow)
    db.session.commit()
    return redirect(url_for('profile', username=username))
```

Finally, let us add this few lines inside index route:

```
    users_followed = Follows.query.filter_by(
        follower=session_user.uid).all()
        uids_followed = [f.following for f in users_followed] +
        [session_user.uid]
      followed_posts = Post.query.filter(Post.author.in_(uids_followed)).all()
```

Migrate Your Local DB to Remote Heroku SQL DB and Post Your App

As a reminder, the steps to migrate your local SQL database to Heroku are described below. (more detailed example here).

1. navigate to your application directory in terminal
2. activate your virtual environment
3. type in terminal

```
    # check if you already have any remote databases created
    Heroku addons

    # make a free heroku database</code>
    heroku addons:create heroku-postgresql:hobby-dev
```

```
# this can view the database you've created
heroku pg:info

# export your postgres path to the current directory export
PATH="/Applications/Postgres.app/Contents/Versions/latest/bin:$PATH"

# push your local database
heroku pg:push your_local_db DATABASE_URL
To connect to your remote database
```

Install flask_heroku: `pip install flask_heroku` and in your *app.py*, instead of:

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://localhost/<dbname>'
```

Use the following:

```
from flask_heroku import Heroku
app = Flask(__name__)
heroku = Heroku(app)
```

In *index.html*, change any links, refs with non-secure connections (`http://`) to secure connections (`https://`) because heroku doesn't allow non-secure connections. And lastly, don't forget to freeze your requirements again, commit and push to Heroku.

Additional Exercises

1. Can you display the username with the post instead of the author id?
2. How might the user be presented with a list of all users who have posted, and choose from that list which one(s) to follow?
3. What happens if you search for a user that doesn't exist? How can this error be prevented?
4. How can you make it so the user can follow or unfollow a user based on a link as well as a button in the form? i.e. , is it possible to just go to the URL '127.0.0.1:5000/unfollow/test_user1' and unfollow a user? How?
5. How would you make a query of all the users *not* being followed?

Extra Credit towards the Midterm (8pts)

Modify Info page. After clicking on “More Details” (Map), users can obtain more information about the specific property (*info.html*). Add new buttons “Save to Favorites” and “Favorites” as presented on the pictures below.

Condo Details

MLS # 72250996

Bedrooms: **2** | Bathrooms: **2.5** | Sqft: **2100** | Price per sqft: **$189.05**

Listing Price: **$409000.00**
Predicted Price: **$407160.88**

> — Lovely townhome in The Preserve a gated community in Hopkinton, starting line of the Boston Marathon. Directly across from Hopkinton State Park with trails & water sports on the lake. Bathed in sunlight with beautiful hardwood floors, classic fireplace in the living room, stainless kitchen and central vac. Two spacious bedrooms on the second floor each with its own bath. Convenient second floor laundry. Lower level is walk-out is nicely finished family/game room and plenty of storage. This beautiful unit is just 7/10 mile, about a 12 minute walk to the commuter rail.

Save to Favorites | Favorites | Home

Condo Details | Your Favorite!

MLS # 72250996

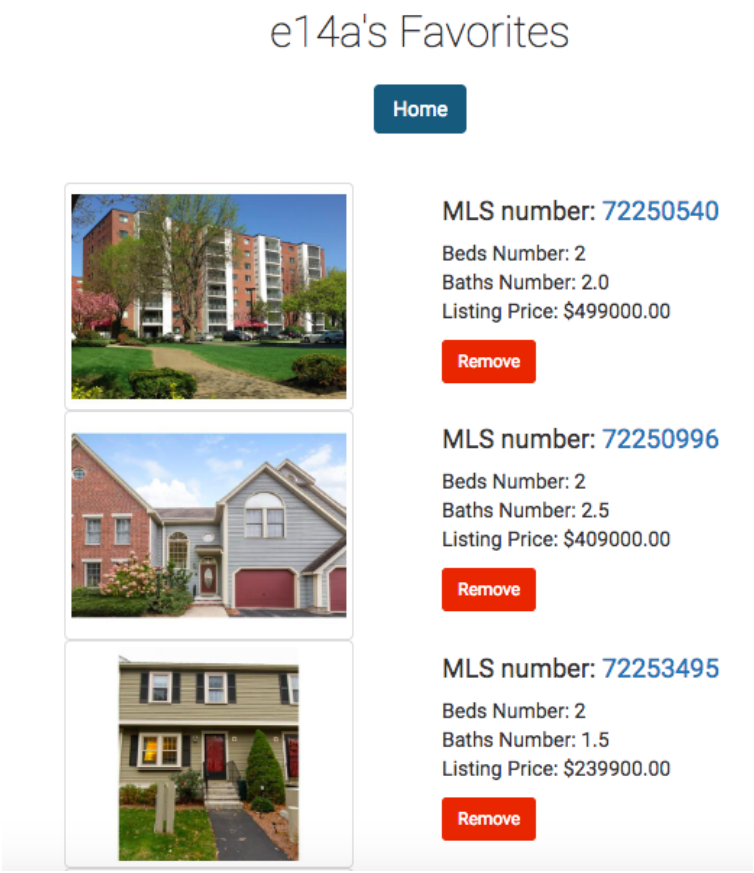
Bedrooms: **2** | Bathrooms: **2.5** | Sqft: **2100** | Price per sqft: **$189.05**

Listing Price: **$409000.00**
Predicted Price: **$407160.88**

> — Lovely townhome in The Preserve a gated community in Hopkinton, starting line of the Boston Marathon. Directly across from Hopkinton State Park with trails & water sports on the lake. Bathed in sunlight with beautiful hardwood floors, classic fireplace in the living room, stainless kitchen and central vac. Two spacious bedrooms on the second floor each with its own bath. Convenient second floor laundry. Lower level is walk-out is nicely finished family/game room and plenty of storage. This beautiful unit is just 7/10 mile, about a 12 minute walk to the commuter rail.

Remove from Favorites | Favorites | Home

Create Profile page. User's profile page (*profile.html*) should contain all user's favorite condos. Please, make sure that information about the property as well as the functionality is the same as presented below:



Credits and Additional Resources

The Flask Mega-Tutorial Part VIII: Followers
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-viii-followers

“Flask Web Development”, 2nd edition by Miguel Grinberg
https://www.oreilly.com/library/view/flask-web-development/9781491991725/